Applying Automated Theorem Proving To Computer Security

THESIS

Kelly McElroy, Capt, USAF

AFIT/GCS/ENG/08-16

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCS/ENG/08-16

Applying Automated Theorem Proving To Computer Security

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

Kelly McElroy, B.S.C.S.

Capt, USAF

March 2008

AFIT/GCS/ENG/08-16

Applying Automated Theorem Proving To Computer Security

Kelly McElroy, B.S.C.S.
Capt, USAF

Approved:

/signed/	3 Mar 2008
Dr. Rusty O. Baldwin (Chairman)	date
/signed/	3 Mar 2008
Lt Col Stuart H. Kurkowski, PhD (Member)	date
/signed/	3 Mar 2008
Dr. Barry E. Mullins (Member)	date
/signed/	3 Mar 2008
Dr. Henry B. Potoczny (Member)	date

Abstract

While more and more data is stored and accessed electronically, better access control methods need to be implemented for computer security. Formal modelling and analysis have been successfully used in certain areas of computer systems, such as verifying the security properties of cryptographic and authentication protocols. However, formal models for computer systems in cyberspace, like networks, have hardly advanced. A highly regarded graduate textbook cites the Take-Grant model created in 1977 as one of the "current" examples of security modelling and analysis techniques. This model is rarely used in practice though.

This research implements the Take-Grant Protection model's four de jure rules and Can_Share predicate in the Prototype Verification System (PVS) which automates model checking and theorem proving. This facilitates the ability to test a given Take-Grant model against many systems which are modelled using digraphs. Two models, one with error checking and one without, are created to implement take-grant rules. The first model that does not have error checking incorporated requires manual error checking. The second model uses recursion to allow for the error checking. The Can_Share theorem requires further development.

Acknowledgements

I would like to thank all my friends for their unwavering support during the last few months, even if their eyes always glazed over after hearing an explanation of my thesis. I would like to thank those special few that actually read my thesis. Special mention goes to:

Fernando for being my rock and being very supportive in and outside the gym. I think he bore the brunt of my thesis problems.

For Bobo–you made me laugh and look at the positive side of the thesis.

For UB, for whom the thesis took time away from and didn't chew up too many things to show his displeasure.

Finally, I would like to acknowledge the great effort of Dr. Baldwin for keeping me on track to finish when there were bumps on the research road.

Kelly McElroy

Table of Contents

Page

List of Figures

List of Lists

List of Abbreviations

Abbreviation		Page
ORNL	Oak Ridge National Laboratory	1
PVS	Prototype Verification System	2
CML	Classical Mathematical Logic	5
FTP	File Transfer Protocol	12
UAC	Unified Access Control	18
TCB	Trusted Computing Base	18
RBAC	Role-Based Access Control	19
ABAC	Attribute Based Access Control	19
TCC	Type Correctness Condition	46

Applying Automated Theorem Proving To Computer Security

I. Introduction

1.1 Background

Formal modelling and analysis have long been successfully used to establish the security properties of cryptographic and authentication protocols [SM93]. Those methods have seen steady progress in the methods and fundamental theories they are based on as well as continued usage demonstrating their value for certain classes of systems. However, formal modelling and analysis of security properties of systems in cyberspace, such as networks, have not significantly advanced since the 1980's [BM07]. The concept of cybercraft or cybertargeting is so recent, very few formal models have been proposed. One paper, "Towards Formal Specification and Verification in Cyberspace" [APN01], which describes a formal model for a simple agent architecture in a multi-agent system offers, a manual algorithm for model checking. Apart from this 2001 model, very few models have been proposed for cybercraft or cybertargeting. However, there are several companies researching how best to apply formal models to secure cyberspace. The Oak Ridge National Laboratory (ORNL), whose mission is to "conduct basic-applied research toward building truly secure, trusted systems beyond best practices," uses formal models to model system security [Oak08].

Nonetheless a highly regarded graduate textbook in this area, [Bis03] for example, uses the Take-Grant model (1977) and the Schematic Protection Model (1988) as "current" examples of security modelling and analysis techniques [BM07]. Even though improvements have been made, these models are rarely used in practice to analyze real systems and no other proposed models have significantly improved the capability to analyze system security properties. The demonstrated usefulness of formal modelling and analysis in the security and safety of protocols and the corre-

sponding slow progress in establishing analogous properties in the cyber arena indicate a new approach may be warranted [BM07].

1.2 Goals of Research

The fundamental goal of this research is to develop a consistent logical framework which allows formal modelling, analysis and reasoning concerning cyberspace security. To do this, the well known and defined Take-Grant Protection model is used as a prototype security policy.

The first goal is to automate the Take-Grant rules, so applying them to realistic problems is easier. The Prototype Verification System (PVS) from SRI [SRI08] uses a combination model checker/theorem prover as a reasoning "engine" and is used to automate this process.

The second goal is to prove the consistency of formal model specifications and prove an application of the rules produces a valid Take-Grant model.

The modelled system is specified as a digraph, with rights and privileges determined by the edges. This follows the original conception of the Take-Grant Model.

Thus, this research produces a Take-Grant protection model to correctly implement a security policy. Performance prediction, as well as identifying vulnerabilities, identifying the operation conditions for those performance predictions can be added to this model.

1.3 Documentation Overview

This document contains five chapters. This chapter gives background on formal modelling and analysis of computer systems. It also defines the purpose of the research to automate the Take-Grant Protection Model in PVS. Chapter 2 reviews current literature and the Take-Grant model as well as current research into the Take-Grant model. Chapter 3 specifies the two models to implement the Take-Grant model and the description of the Can_Share predicate algorithm. Chapter 4 presents the proofs

for the two specification models and the code to implement the Can_Share algorithm. Chapter 5 contains the conclusions of the research and identifies future research areas.

II. Automated Theorem Proving to Improve Computer Security

2.1 Introduction

This chapter covers the background of computer security and formal models. Also the current research into the Take-Grant Protection model is discussed, which is the focus of this research. Finally current research into formal models for access control is presented.

2.2 Background

Computer security is increasingly important as more of our data are stored and accessed electronically, which means better access control methods need to be implemented. Access control protects resources by explicitly enabling or restricting the use of that resource [Nat96]. Access is generally based on a security policy which specifies what is or is not allowed. Computer-based access controls are called logical access controls [Nat96]. Formal models of access control policies are usually based on propositional and predicate logic, otherwise known as classical logic, where the "truth" about some aspect of the system is reduced to TRUE, FALSE, or UNDECIDABLE. That is, a property holds (is TRUE), does not hold (is FALSE), or it cannot be determined if it holds (is UNDECIDABLE) [BM07]. "[T]o specify, verify, and reason about information security and information assurance, we need a right fundamental logic system to provide us with a logical validity criterion of normative reasoning as well as formal representation and specification language [CM06]." Fundamental logic must support truth-preserving and relevant reasoning in the sense of the conditional, ampliative reasoning, semi-complete reasoning, semi-consistent reasoning, and normative reasoning. The essential requirements to support fundamental logic are [CM06]:

- relevant reasoning - the premise must be relevant to the conclusion and vice versa.

- truth-preserving reasoning - an argument is valid if and only if it is impossible for all its premises to be true while its conclusion is false in the sense of the conditional.
- ampliative reasoning - the truth of the conclusion of the reasoning should be recognized after the completion of the reasoning process but not be invoked in deciding the truth of premises of the reasoning.
- semi-complete and semi-consistent reasoning - understanding that the current knowledge may be incomplete and or inconsistent in many ways, there is no evidence for deciding the truth of either a proposition or its negation and or whether it directly or indirectly includes contradictions.
- normative reasoning - often describes only ideal situations, even when they are used in actual situations, therefore logic must be able to distinguish between what ought to be done and what is the case.

Classical mathematical logic (CML) cannot satisfy any of the fundamental logic criteria because they are not necessarily relevant, truth-preserving in the sense of the conditional, it is circular, not ampliative, and reasoning under inconsistency is impossible. Furthermore, since relevance between the premises and conclusion of an argument is not accounted for by the classical validity criterion in CML, reasoning based on CML is not necessarily relevant [CM06]. Therefore, reasoning about systems with access control properties requires a different logic system.

Modal logic broadly defines a family of logics that capture "modes" of truth. Modal logic considers not only truth and falsity applied to what is or is not so as things actually stand, but considers what would be so if things were different, i.e., modal logic is concerned with truth or falsity in possible worlds as well as the real one [HC96]. A mode is an expression used to qualify the truth of a judgment, generally based on the morals applied to it [Gar07]. The best-known logics of modal logic are Modal, Deontic, Temporal, Epistemic, and Relevance. According to [Gar07]:

- Modal logic is the logic of necessity and possibility.

- Deontic logic is the logic of obligation, permission, and forbidden.
- Temporal logic is the logic of time-"it will be the case", "it will always be the case", "it has always been the case", and "it was the case".
- Epistemic logic is the logic of belief and knowledge.

Relevance logic reasons about systems that have contradictory premises. Modal logics have similar rules and a variety of symbols. They are more expressive in their expressive and reasoning ability in analysis. However, the question still remains whether they are more suitable than classical logic to accurately model protection schemes.

Even though modal logics show promise, fundamental research automating the analysis of protection schemes based on CML still need a thorough examination as well.

"One of the most critical and least understood aspects of protection is the exercise of control over the movement of rights between subjects of a system [LM82]." An access control scheme specifies the state-transition rules of a system based on that scheme; a set of them is an access control model [TL04]. A well-known access control model is the Take-Grant Protection Model. The Take-Grant Protection model has two components: "a finite, labelled, directed two color graph representing the protection state of an operating system and a finite set of graph transformation rules with which the protection state may be changed [Sny81]." The distribution of rights among the various subjects of a system at a given moment is called the protection state of the system [LM82]. The Take-Grant model is made up of subjects and objects that have rights. Only subjects can use rights to invoke protection system rules to change the protection state of the system. Objects may hold rights, but cannot invoke rules. The Take-Grant model has four rewriting rules called de jure rules: Take $\equiv t$,
Grant $\equiv g$, Create and Remove, which changes the protection state. The Take rule allows the subject invoking it to obtain the rights of another subject or object it has take rights over. The Grant rule allows a subject to grant rights it possesses to another

subject or object, assuming it has grant rights over that subject or object. Create forms a new subject/object and the Remove rule removes rights to a subject/object.

The following rule descriptions are due to [Bis96], and demonstrate the de jure rules. Consider a directed protection graph in which the labelled edges represent rights and vertices represent entities. Entities are either subjects, represented by ●, or objects, represented by ◯. Vertices that could be either subjects or objects are represented by ⊗. The $\vdash$ symbol denotes the graph G_1 can be derived from G_0 in one step.

An example of the Take rule is shown in Figure 2.1. In the protection graph G_0, let **x**, **y**, **z** be three distinct vertices, where **x** is a subject. Let there be an edge from **x** to **y** labelled t, an edge from **y** to **z** labelled β, and $\alpha \subseteq \beta$, where β is the set of rights **y** possesses over **z** and α is a subset of those rights. Then the Take rule defines a new graph G_1 by adding an edge to the protection graph from **x** to **z** labelled α. The rule is written "**x** takes (α to **z**) from **y** [Bis96]."

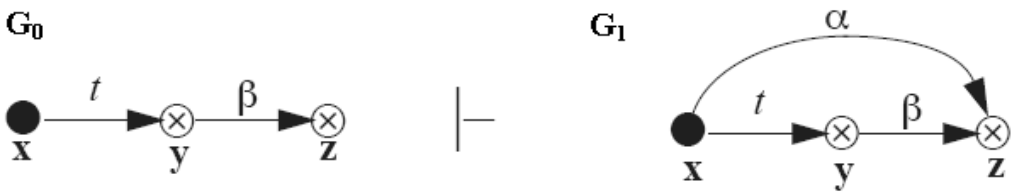

Figure 2.1: Take Rule [Bis96]

The Grant rule is shown in Figure 2.2 [Bis96]. As before, **x**, **y**, **z** are distinct vertices where **x** is a subject. Let there be an edge from **x** to **y** labelled g, an edge from **x** to **z** labelled β, and $\alpha \subseteq \beta$. The Grant rule defines a new graph G_1 by adding an edge to the protection graph from **y** to **z** labelled α. The rule is written "**x** grants (α to **z**) to **y** [Bis96]."

The Create rule is shown in Figure 2.3 [Bis96]. In protection graph G_0, **x** is a subject and $\alpha \subseteq R$, where R is the set of all rights defined for this system. The Create

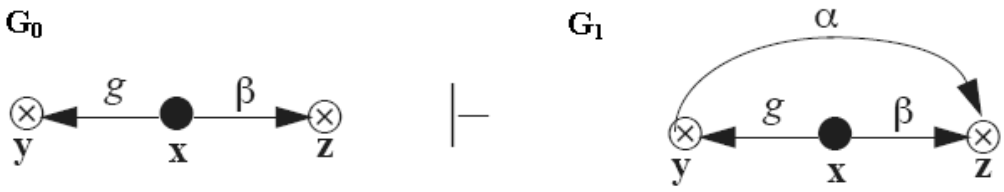

Figure 2.2: Grant Rule [Bis96]

rule defines a new graph G_1 by adding a new vertex **y** to the graph and an edge from **x** to **y** labelled α. The rule is written "**x** creates (α to a new vertex) **y** [Bis96]."

Figure 2.3: Create Rule [Bis96]

An example of the Remove rule is shown in Figure 2.4 [Bis96]. Let there be an explicit edge from **x** to **y** labelled β with $\alpha \subseteq \beta$. Then Remove defines a new graph G_1 by deleting the α labels to β. If β becomes empty as a result, the edge itself is deleted. The rule is written "**x** removes (α to) **y** [Bis96]." The Take-Grant

Figure 2.4: Remove Rule [Bis96]

model can determine the safety of a specific system in linear time [Bis03]. A policy is considered safe if a system never has unauthorized transfers of rights. Safety refers to the abstract model while security refers to the implementation of the model. The reason for the two different terms is that a safe system can be made unsecure through implementation. The terms used specifies where the error lies [Bis03].

2.3 Current Take-Grant Protection Model Research

The Take-Grant Protection Model has been extended in many ways, but has yet to have modal logic applied to it. After the Take-Grant Protection Model was introduced in 1976 the first modifications to it was by [BS79]. Transfer methods called de jure and de facto were identified. The name de jure was given to the operations already defined: Take, Grant, Create, Remove and the model was extended to include de facto or implicit transfers in the operations: Post, Pass, Spy, and Find.

De jure captures the direct authority to read information while de facto transfers refer to a user acquiring the information without getting direct authority to read it [BS79]. The de jure rules change the protection state of the graph while the de facto rules, model what happens when operations occur but do not change the graph state [FB96]. The following examples of the de facto rules: Post, Pass, Spy, and Find are from [Bis96].

An example of the Post rule is shown in Figure 2.5. [Bis96]

Figure 2.5: Post Rule [Bis96]

In a protection graph on the left, let there be an edge from **x** to **y** labelled r, an edge from **z** to **y** labelled w. The Post rule defines the graph on the right with an implicit edge from **x** to **z** labelled r. The rule is then written "**z** posts to **x** through **y** [Bis96]."

An example of the Pass rule is shown in Figure 2.6 There is an edge from **y** to **x** labelled w and an edge from **z** to **y** labelled r. The Pass rule defines the new graph on the right with an implicit edge from **x** to **z** labelled r. The rule is then written "**y** passes from **z** through **x** [Bis96]".

Figure 2.6: Pass Rule [Bis96]

The Spy rule is shown in Figure 2.7 [Bis96]. In the left protection graph there is an edge from **x** to **y** labelled r, and an edge from **y** to **z** labelled r. The Spy rule defines the graph on the right with an implicit edge from **x** to **z** labelled r. The rule is written "**x** spies on **z** using **y** [Bis96]."

Figure 2.7: Spy Rule [Bis96]

The Find rule is shown in Figure 2.8 [Bis96]. An edge from **y** to **x** labelled w, an edge from **z** to **y** labelled w results in a "find" which the new graph on the right shows with an implicit edge from **x** to **z** labelled r. The rule is written "**x** finds from **z** through **y** [Bis96]."

Figure 2.8: Find Rule [Bis96]

The predicate Can_Know uses the de facto rules, while the predicate Can_Share uses the de jure rules [BS79]. The Can_Know predicate is true when an implicit edge

can be added by means of using the de facto rules. The *Can_Know* is the de facto version of *Can_Share* [Bis95]. The *Can_Share* predicate is true when an edge can be explicitly added by means of using the de jure rules. In [Sny81], the predicate *Can_Steal* captures "the notion that a subject vertex acquires a new right without cooperation from an original owner" while the *Can_Share* predicate assumes cooperation from all users [Sny81].

[Bis81] applies the Take-Grant Protection Model to a hierarchical protection system with the focus on transfer of information and authority instead of rights. Conditions are established to make the hierarchical system secure no matter how many of its subjects are corrupt. The model was developed under the assumption that no user should be able to break security, an assumption not present in earlier Take-Grant models of hierarchical protection systems.

[LM82] recognized that the Take-Grant Protection Model could not enforce unidirectional channels–information was free to flow from one subject to another, either directly or indirectly. Because the unidirectional transfer of rights limits the applicability of the model, [LM82] extended it to include Take-Receive which limits the flow of rights. However, because Take-Receive method is a simplification of the "Send-Receive" transport control mechanism used in the Operation Control protection scheme [Min78], it was not proposed as a control mechanism for a real system but to demonstrate how to avoid necessarily symmetric flows of rights in a real system [LM82].

Bishop combines the notion of theft with the notion of information flow between two objects extending the idea of conspiracy to the theft of information, using the de facto rules [Bis95]. Most other papers referenced are concerned with subject only "thefts." A new predicate *Can_Snoop* is the de facto version of *Can_Steal*. *Can_Snoop* is true when there is no cooperation on the part of the snoopee with the snooper. With this extension the Take-Grant Model can model very practical concepts and is no longer simply a theoretical tool.

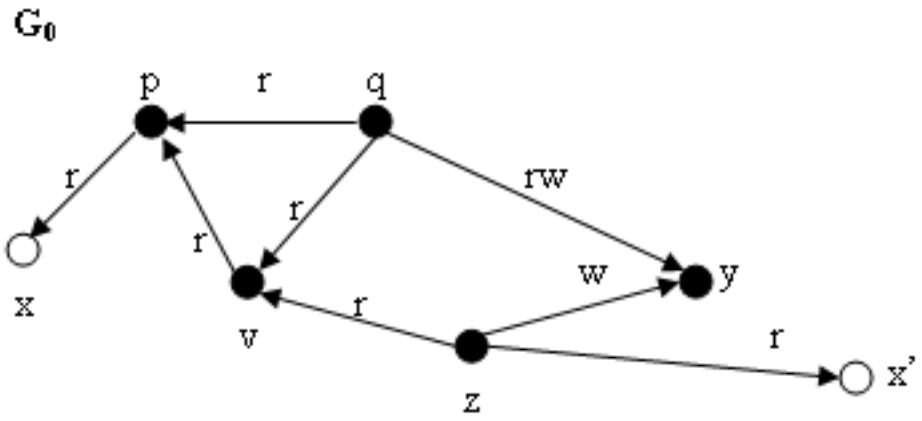

Figure 2.9: Conspirators in an information flow modified from [Bis96]

The idea of "conspirators" has also been extended to information flow [Bis96]. The precise bound on the number of actors required for information to be transferred from one vertex to another are established. Bishop demonstrated conspirators to information flow in a small local network using the File Transfer Protocol (FTP). A simplified example follows. Using the network configuration in Figure 2.9, the number of conspirators needed to make a copy of the file **x** and place it on **z** is determined. All information transfers are along implicit edges which needs the following abstract representation [Bis96]: subjects represent hosts, objects represent files, and permission for an entity to retrieve or access files is represented by an explicit read (r) edge from either host to host or host to file. There are five hosts: **p, q, y, v,** and **z**.

To find the number of conspirators needed for the original protection, G_0, graph shown in Figure 2.9 a corresponding acting graph is developed. This graph consists of vertices corresponding to access sets in the original graph G_0 with edges corresponding to paths along which the focus of each access set can pass information acting alone [Bis96].

Given a protection graph G_0 with subject vertices p, q, v, z and y, an acting graph G_1, Figure 2.10, is generated with vertices p, q, v, z and y. Each vertex in G_1 is

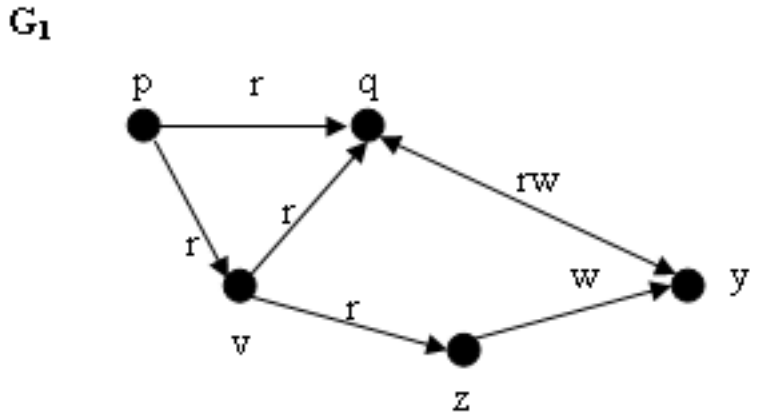

Figure 2.10: The corresponding acting graph

associated with the access sets $I(y)$ and $T(y)$ from G_0 where $I(y)$ is an initial access set containing y and all vertices to which y initially or rw-initially spans [Bis96]. $T(y)$ is the terminal access set containing y and all vertices to which y terminally or rw-terminally spans [Bis96]. An initial span of a subject v_0 is a tg-path between v_0 and v_n with an associated word in $\{\overrightarrow{t}^*\overrightarrow{g} \bigcup \nu\}$, where ν is a null-span, and rw-initially spans if there is an rwtg-path between v_0 and v_n with associated word in $\{\overrightarrow{t}^*\overrightarrow{w}\} \bigcup \{\nu\}$ [Bis96]. A terminal span of a subject v_0 is a tg-path between v_0 and v_n with an associated word in $\{\overrightarrow{t}^*\}$ and rw-terminally spans if there is an rwtg-path between v_0 and v_n with associated word in $\{\overrightarrow{t}^*\overrightarrow{r}\}$ [Bis96].

To determine the acting graph G_1, the access sets have to be built for G_0, which are:

$I(p) = \{p\}$ $T(p) = \{x\}$ $I(z) = \{z, y\}$ $T(z) = \{z, v, x'\}$ $I(q) = \{q, y\}$
$T(q) = \{p, q, y\}$ $I(v) = \{v\}$ $T(v) = \{p, v\}$ $I(y) = \{y\}$ $T(y) = \{y\}$

$T(y)$ is the maximal set of vertices from which y can obtain information, and $I(y)$ is the maximal set of vertices to which y can pass rights or information. These sets are not necessarily identical and this adds significant complexity to the conspiracy problem [Bis96]. Next the sets $\triangle(a, b)$ for each pair of vertices a and b are built from G_0 access sets. The set $\triangle(a, b)$ is defined to be all vertices in $I(a) \bigcap T(b)$ except

those vertices y which are information gates (i.e., $T(x) = I(x) = x$). This means the set $\triangle$ includes only those vertices to which the foci can pass (or receive) information with the foci being the only actors. The non-empty sets are

$$\triangle(p,q) = \{p\}, \quad \triangle(p,v) = \{p\}, \quad \triangle(q,y) = \{y\}, \quad \triangle(z,q) = \{y\},$$
$$\triangle(z,y) = \{y\}, \quad \triangle(v,z) = \{v\}, \quad \triangle(y,q) = \{y\}$$

From these sets the acting graph, G_1, is built and is shown in Figure 2.10. Consider the information flow from x to x' in Figure 2.9. In this case, $I_x = \{p\}$ and $T'_x = \{z\}$. The only path between p and z has three vertices (p, v, and z) in Figure 2.10, so the minimum number of actors necessary and sufficient to move the information from x to x' is 3 (with p, v, and u being the three actors) [Bis96].

A key factor in information security is the ability to identify useable metrics to measure the strength of a security policy [Wan05]. One metric that can be determined in the Take-Grant model is the amount of cooperation required to share or steal rights. This is the number of users required to initiate rules for a particular edge to be added to a graph.

An example follows starting with $\mathbf{G}_0$ shown in Figure 2.11 to demonstrate how many conspirators are needed to witness **Can_Share(r,x,y,$\mathbf{G}_0$)**, which is read 'can subject **x** obtain **r** (read) rights over subject **y**'? The example is modified from [Bis03].

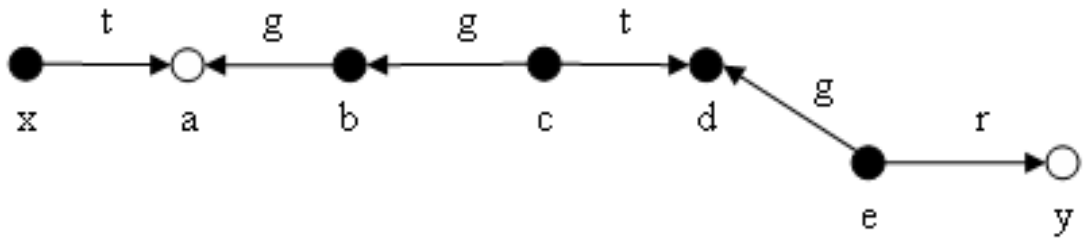

Figure 2.11: A Take-Grant Protection Graph: G_0

The first conspirator is **e** which grants (**r** to **y**) to **d**, as shown in Figure 2.12.

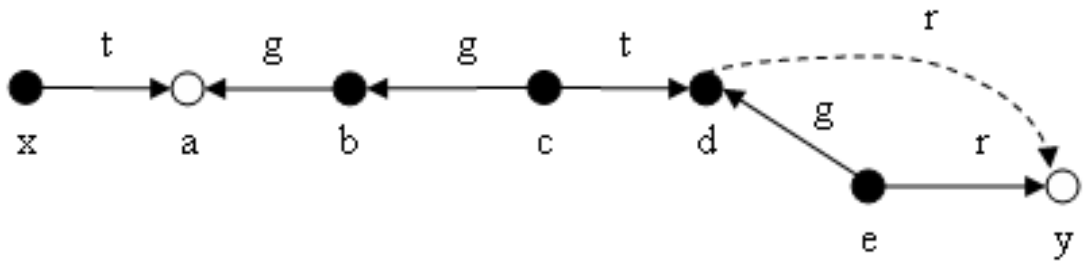

Figure 2.12: First conspirator: e

The second conspirator, **c**, takes **r** to **y** from **d**, as shown in Figure 2.13.

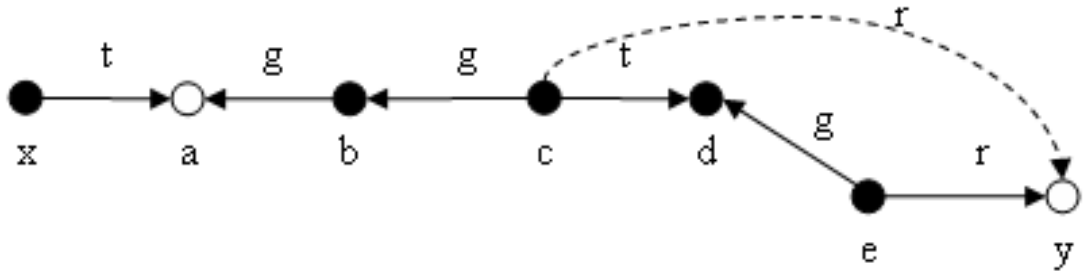

Figure 2.13: Second conspirator: c

Then **c** grants (**r** to **y**) to **b**, as shown in Figure 2.14.

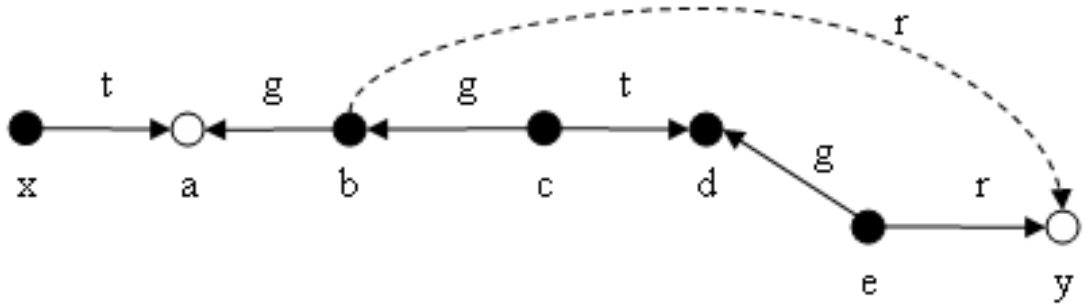

Figure 2.14: C conspires twice

The third conspirator is **b** which grants (**r** to **y**) to **a**, as shown in Figure 2.15, and

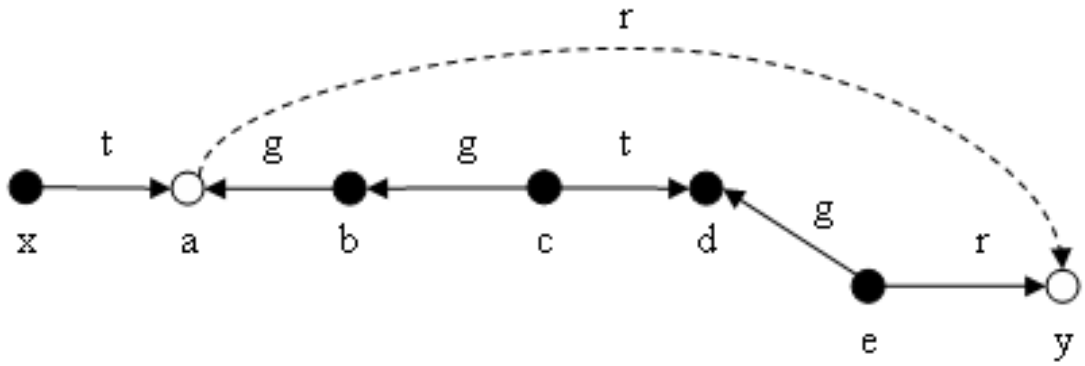

Figure 2.15: Third conspirator: *b*

finally, the last conspirator **x** takes (**r** to **y**) from **a**, is shown in Figure 2.16

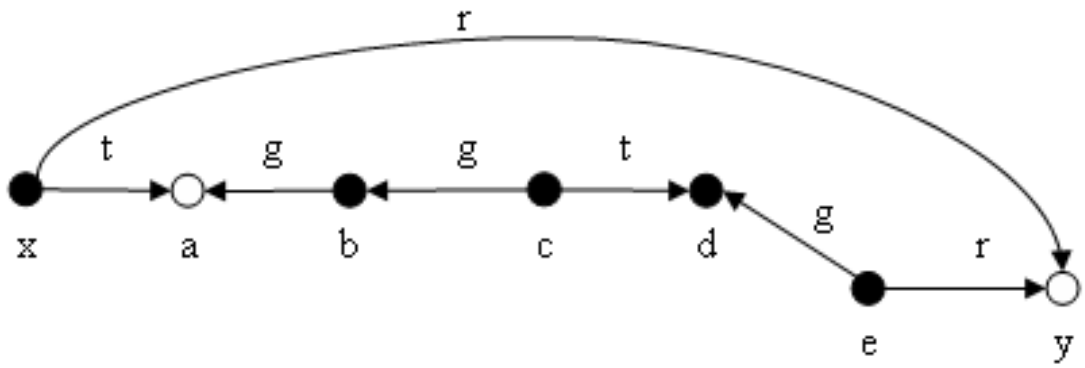

Figure 2.16: Conspiracy Graph: $\mathbf{G}_1$, with **x** as the last conspirator

[Sny81] derived exact conspiracy measurements for arbitrary protection graphs and presented the first algorithm for discovering minimum conspiracy. This conspiracy graph can be evaluated in linear-time, and requires n^2 operations for an n subject graph to fill the edges.

The Take-Grant model has also been represented as a Petri net [Mar93]. A Petri net is a "graph with two types of nodes (bipartite graphs), transitions and places, where the arcs connect either transitions to places or places to transitions [JMS06]."

By implementing the Take-Grant model in a Petri net it is shown there is an efficient algorithm to determine the cooperation required to share or steal rights in linear time. The algorithm found all rights a subject can steal with the help of a given set of conspirators to be 3n + O(L) where L is the number of labels Take $\equiv t$ and Grant $\equiv g$ in the initial protection graph [Mar93].

2.4 Current Research

The demonstrated usefulness of formal modelling and analysis in the security and safety of protocols and the corresponding lack of progress in establishing analogous properties in the cyber arena indicate a new approach may be warranted [BM07].

A protection model that can verify whether a given protection scheme correctly implements a security policy, performance prediction, as well as identifying vulnerabilities, identifying the operation condition assumptions for those performance predictions may require a form of modal logic be applied to overcome the deficiencies of classical mathematical logic.

To implement access control, a security policy must be defined. Leiwo [LZ97] and Cuppens [CS96] use formal models to define security policies. Leiwo's [LZ97] formal model documents and standardize information security requirements by dividing security into objectives, strategies, and policies that are refined into concrete protection measure specifications. "The model assumes a hierarchical, layered, information security development organization and specifies vertical and horizontal harmonization functions in order to establish cost effective protection [LZ97]." The vertical dimension provides each layer a common view of requirements. The horizontal dimension identifies similar security requirements at each layer which simplifies their implementation and maintenance. This approach starts with a formal model that incorporates the total security requirements to establish cost effective protection. Whereas Leiwo [LZ97] focuses on defining the security policy, Cuppens [CS96] focuses on the formalization of security policies with language specifications.

Cuppens scheme is domain-independent and reusable. Consider a generic corporate multilevel security policy whose objective is to determine if the policy system conforms to a specific set of regulations. A regulation may be viewed as being composed of agents, events, and objects of the system to be regulated. Regulations defining what actions of the agents are permitted, obliged or prohibited to do [CS96]. Two classes of constraints are identified: agent enforced constraints on actions on system objects and agent enforced constraints on interaction with other agents [CS96]. To describe the regulations, a logic based approach is used that incorporates deontic and temporal modal logic concepts with organizational concepts of responsibility, delegation, actions and events. For example, "Every organization which holds some secret documents is **obliged** [deontic] to designate an agent who is **responsible** [organizational] for preserving these documents" and "**Before** [temporal] every **meeting** [event], the organizer of this meeting is **obliged** [deontic] to **establish** [action] a list of all participants in this meeting [CS96]". Violations of policy can also be specified in this model.

La Padula [Pad90] discusses a domain-independent formal model which implements a new approach to computer security using the "Unified Access Control" (UAC) framework developed at MITRE. UAC permits flexibility in choosing and specifying security policies for a secure system because all access control is based on a small set of fundamental concepts. The UAC framework is generalized for "computer access control and explicitly recognizes the fundamental components for access control-attributes, rules, and authority, and relates them in a utilitarian manner [Pad90]." The only difference in the formal model (which is derived from the UAC framework) compared to traditional models is that the rules for access control are a separate entity from the model of the Trusted Computing Base (TCB) where certain subjects, objects, processes, are exempt from the security policy so as to carry out their functions [Pad90]. The model does not encumber rules with the details of an actual system, allows the reconfiguration of security policies without reevaluating assurance, and has flexibility in implementation.

The security policy should describe the type of access control to be implemented in the system. Kolaczek [Kol02] realizes the limitations of traditional access control and applies deontic logic to role-based access control (RBAC), which is based on the job a user holds, not the user's identity. Typical RBAC would simply either allow access or not. However, in reality that is generally not how access is qualified, which the use of deontic logic allows. Attribute based access control (ABAC), grants access control based on attributes associated with the user and typically uses temporal and deontic logic. However, most security policies focus on access control which does not necessarily protect the flow of information.

Sabelfield [SM03] compared three decades of research on information-flow security, focusing mostly on work that uses static program analysis to enforce information-flow policies. He concludes that current security measures such as access control and encryption do not restrict information flow and neither do language-based techniques, in particular, program semantics and analysis for the specification and enforcement of security policies for data confidentiality [SM03].

Thus information flow along with access control are both needed. The Take-Grant model, with de jure and de facto rules, can support a formal model that considers both those needs. Current research suggests applying modal logic to formal models may be necessary to significantly advance formal security modelling capabilities.

2.5 Summary

This chapter covered formal modelling and why it is necessary for security policies, which essentially determines access control for a system. The Take-Grant Protection model rules used in this research were also discussed.

Current research focuses on the methods of access control, that do not protect information flow. However, the Take-Grant Protection model has rules to determine information flow along with access control, thereby adding extra validity for it as a security policy prototype.

III. Specifying the Take-Grant Model in PVS

3.1 Introduction

"Specification language, theorem provers, and models checkers are beginning to be used routinely in industry [HR04]." This research specifies the Take-Grant Protection Model in the Prototype Verification System (PVS), a well-known verification system [For03]. PVS uses a specification language to express mathematical theorems and conjectures based on a system definition, which can be discharged using the interactive theorem prover. The specification language has a rich and expressive logic, and can formulate and examine problems in computer science; however, it does not provide built-in notions of "state," "state variable," or "variable assignment" [For03]. These computational notions have to be modelled explicitly by encoding the semantics of state machines and their transitional relations within the specifications. The PVS theorem prover is interactive and based on a sequent calculus. One of the main uses of PVS has been to explore synergies between expressive logic and proof automation [For03]. Because PVS is free of specific notions of computation, it is an ideal platform to examine the Take-Grant Protection Model. Using PVS, the behavior of Take-Grant is validated for a specified graph.

Two separate specifications are developed for each of the Take-Grant rules: Take, Grant, Create, and Remove. Both use a digraph for model entities. Both specifications are subject only. The first specification is based on the Take-Grant theorem directly. The second uses recursion and case statements to check the specified Take-Grant graph for validity. Because the Take-Grant rules have common functions, code for each rule was divided into parameterized theory modules and imported into the rule theories. This chapter is composed of two model specification sections, with the common imported theories and rule theories making up subsections under those. The final two sections present the Can_Share section and a section discussing contributions made to the PVS digraph library.

3.2 Take-Grant Model One - No Error Checking

3.2.1 Common Imported Theories. In PVS, a specification consists of a collection of theories, which may contain the type names and constants, axioms, definitions, and theorems associated with the specification [OSRSC99a]. Theories are imported by theory name and not by file name. Different files having the same theory name will cause errors. To avoid duplication, seven theories are developed that can be imported to use with the Take-Grant rules depending on the specification. Those theories are Definitions, Add_Edge, Take_Graph_Init, Grant_Graph_Init, RemoveEdge, TG_Lemma_Init_Take, and TG_Lemma_Init_Grant.

Definitions Theory

This theory in List 3.1 contains all the declared types and constants, along with a single axiom. This theory is used in all four rules.

List 3.1: Definitions Theory

```
%Part 1:
Definitions[Vertex:Type+]:THEORY
    % Definition file: used with all the rules
BEGIN

%Part 2:
Importing digraphs@digraphs[Vertex]

%Part 3: declares TYPE of rights
Rights: TYPE = {read, write, take, grant}

%Part 4: declares a function given an edge returns the rights that...
    belong to it
E_DB: TYPE =function[edgetype[Vertex]->set[Rights]]

%Part 5: Vertex's that can be used in the graph
X,Y,Z,A: Vertex
```

```
%Part 6: Variable that contains all rights
all_rights: Rights

%Part 7:  AXIOM: which states the vertex aren't equal.
    not_eq_ax: AXIOM X/=Y and Y/=Z and Z/=X and X/=A and A/=Y and ...
       A/=Z

%Part 8:
END Definitions
```

Part 1 names the theory and declares a non-empty parameter type Vertex. Because the Take-Grant rules are specified using a graph, each subject is of type Vertex. *BEGIN* declares the beginning of the theory.

Part 2 imports the digraph theory and all diagraph functions instantiated for the Vertex type. Digraph was developed by NASA Langley.

Part 3 declares an enumerated type called Rights with elements: read, write, take, or grant.

Part 4 declares a type E_DB which serves as an edge database. E_DB is a function which returns the set of rights for a given edge. For instance, if an edgetype edge (X, Y), where X and Y are type Vertex, is initialized with read and take rights, **E_DB(edge(X, Y))** would return read, take.

Part 5 declares all the four graph vertices as constants.

Part 6 declares all variable all_rights to contain the four rights read, write, take, grant.

Part 7 defines the axiom not_eq_ax used in the Take-Grant specifications. This axiom states that all the vertices are different.

Part 8 ends the Definitions Theory with the keyword *END* and the theory name.

Add_Edge Theory

List 3.2, Add_Edge, contains all the functions to add edges and vertices to the graph, along with assigning rights to edges. Add_Edge is used by the Take, Grant and Create rules. In the following, only those aspects of the PVS specification not previously discussed is covered.

List 3.2: Add_Edge Theory

```
%Part1:
Add_Edge  [Vertex:Type+]: THEORY BEGIN

%Part 2:
Importing Definitions[Vertex]

%Part 3: Adds a new edge to graph
AddEdge(g1: digraph, x: Vertex, y: Vertex):
    digraph[Vertex] =
        if vert(g1)(x) and vert(g1)(y)
        then g1 with [edges:= add((x,y),edges(g1))]
        else g1
        endif

%Part 4: Adds a right to the new edge
AddEdgeRight(db: E_DB, x: Vertex, y: Vertex, r: Rights):
    E_DB = db with [(x,y):= add(r, emptyset[Rights])]

%Part 5: Adds all rights to the new edge
AddEdgeAllRights(db: E_DB, x: Vertex, y: Vertex):
    E_DB = db with [(x,y):=
    add(read,add(write, add(take, add(grant, emptyset[Rights]))))]

%Part 6:  For Adding Verts
AddVert(g1:digraph, x:Vertex):
    digraph[Vertex] =
        if vert(g1)(x)
```

```
        then g1
        else g1 with [vert:= add(x,vert(g1))]
        endif

%Part 7:
END Add_Edge
```

The *AddEdge* function in Part 3 adds a new edge to the digraph. The function takes a digraph and two vertices. If both vertices exist in the graph, the new edge is returned, otherwise the graph is returned unchanged.

In Part 4, *AddEdgeRight* takes an edge database, E_DB, and two vertices. The two vertices define the edge that rights are assigned to. E_DB is returned with updates to the specified edge.

The *AddEdgeAllRights* function in Part 5 takes an edge database E_DB, two vertices, and rights and returns E_DB with the specified edge containing all rights.

In Part 6, the *AddVert* function is used for the Create Rule. It takes a digraph and a vertex. If the vertex exists, the digraph is returned with no changes, if it does not exist then the vertex is added to the graph and the updated digraph is returned.

Graph_Init Theory

This theory shown in List 3.3 initializes the digraph and the edge database for the Take rule. It is used by the Take, Create, and Remove rules. Create and Remove digraphs can be initialized with any configuration.

List 3.3: Graph_Init Theory

```
%Part 1:
Graph_Init  [Vertex:Type+]: THEORY
%Initializes the graph for the Take Rule
BEGIN
%Part 2:
Importing Definitions[Vertex]
```

```
%Part 3: Initialize graph
InitGraph: digraph[Vertex] =
    (#vert:= add(X, add(Y, singleton[Vertex](Z))),
    edges:= add((X,Y),add((Y,Z),emptyset[edgetype]))#)

%Part 4: initializes the edge rights
ADD(db: E_DB): E_DB = db
    with [(X,Y):= add(take, emptyset[Rights])]
    with [(Y,Z):= add(read,emptyset[Rights])]

%Part 5:
END Graph_Init
```

Part 3 *InitGraph* initializes the digraph for the Take rule by first adding the vertices X, Y and Z to the graph and the edges (X, Y) and (Y, Z).

In part 4, *ADD* initializes the right database for the graph declared in *InitGraph*. To the edge (X, Y) the take right is added and to the edge (Y, Z) the read right is added.

TG_Lemma_Init_Take Theory

In List 3.4 the theory initializes the digraph and the edge database for the Take-Grant Take lemma. It is used only by the Take rule. This theory is similar to the Grant_Graph_Init theory, List 3.5, except the edge right is a take instead of a grant.

List 3.4: TG_Lemma_Init_Take Theory

```
% Part 1:
TG_Lemma_Init_Take[Vertex:Type+]: THEORY
%Initializes the graph for the Take Take-Grant Lemma
BEGIN

%Part 2:
Importing Definitions[Vertex]
```

```
%InitGraphL and ADDL used for Take-Grant Lemma
%Part 3: Initialize graph
InitGraphL: digraph[Vertex] =
    (#vert:= add(X, add(Y, singleton[Vertex](Z))),
    edges:= add((Y,X), add((Y,Z),emptyset[edgetype]))#)

%Part 4: initializes the edge rights
ADDL(db: E_DB): E_DB = db with [(Y,X):=
    add(take, emptyset[Rights])] with [(Y,Z):= add(read,emptyset[...
       Rights])]

%Part 5:
END TG_Lemma_Init_Take
```

In part 3, *InitGraphL*, initializes the digraph for the Take-Grant Take lemma and adds the vertices X, Y and Z to the graph. And the edges (Y, X) and (Y, Z).

ADDL, in part 4, initializes the right database for the graph declared in *InitGraphL*. The take right is added to the edge (Y,X) and read is added to the edge (Y, Z).

Grant_Graph_Init Theory

This theory initializes the digraph and the edge database for the Grant rule in List 3.5. It is used only by the Grant rule for this specification.

List 3.5: Grant_Graph_Init Theory

```
%Part 1:
Grant_Graph_Init  [Vertex:Type+]: THEORY
%Initializes the graph for the Grant rule
BEGIN

%Part 2:
Importing Definitions[Vertex]
```

```
%Part 3: Initialize graph
InitGraph: digraph[Vertex] =
    (#vert:= add(X, add(Y, singleton[Vertex](Z))),
      edges:= add((Y,X),add((Y,Z),emptyset[edgetype]))#)

%Part 4: initializes the edge rights
ADD(db: E_DB): E_DB = db
    with [(Y,X):= add(grant,emptyset[Rights])]
    with [(Y,Z):= add(read,emptyset[Rights])]

%Part 5:
END Grant_Graph_Init
```

Part 3, *InitGraph*, initializes the digraph for the Take-Grant Take lemma. It adds the vertices X, Y and Z to the graph and edges (Y, X) and (Y, Z).

ADD in Part 4 initializes the rights database for the graph declared in *InitGraph*. To the edge (Y, X) grant is added and read is added to the edge (Y, Z).

TG_Lemma_Init_Grant Theory

This theory initializes the digraph and the edge database in List 3.6 for the Take-Grant Grant lemma. It is used only by the Grant rule in this specification.

List 3.6: TG_Lemma_Init_Grant Theory

```
%Part 1:
TG_Lemma_Init_Grant [Vertex:Type+]: THEORY
%Initializes the graph for the Grant Take-Grant Lemma
BEGIN

%Part 2:
Importing Definitions[Vertex]

%InitGraphL and ADDL used for Take-Grant Lemma
```

```
%Part 3: Initialize graph
InitGraphL: digraph[Vertex] =
    (#vert:= add(X, add(Y, singleton[Vertex](Z))),
    edges:= add((X, Y), add((Y,Z),emptyset[edgetype]))#)

%Part 4: initializes the edge rights
ADDL(db: E_DB): E_DB = db
    with [(X,Y):= add(grant,emptyset[Rights])]
    with [(Y,Z):= add(read,emptyset[Rights])]

%Part 5:
END TG_Lemma_Init_Grant
```

Part 3 *InitGraphL* initializes the digraph for the Take rule. It adds the vertices X, Y and Z to the graph. And the edges (X, Y) and (Y, Z).

Part 4 *ADDL* initializes the right database for the graph declared in *InitGraphL*. To the edge (X, Y) the grant right is added, and to the edge (Y, Z) the read right is added.

Remove Edge Theory

This theory, List 3.7, contains all the functions to remove edges from the digraph, along with the rights. It is used only by the Remove rule in this specification.

List 3.7: Remove Edge Theory

```
%Part 1:
RemoveEdge [ Vertex: Type+]: THEORY
%Used by Remove Right
BEGIN

%Part 2:
Importing Definitions[Vertex]
```

```
%Part 3:    Removes edges
RemoveEdge(g1: digraph, e:edgetype):
    digraph[Vertex] =  g1 with [edges:= remove(e,edges(g1))]

%Part 4:    Removes rights to an edge
RemoveEdgeRight(db: E_DB, e:edgetype, r: Rights):
    E_DB = db with [(e`1,e`2):= remove(r, db(e))]

%Part 5:
END RemoveEdge
```

The *RemoveEdge* function, Part 3, takes a digraph and an edgetype, which is an edge in the digraph, and returns the digraph after the edge is removed from the digraph's set of edges.

In part 4, *RemoveEdgeRight*, takes the edge database, an edgetype, and a right and returns the edge database with the right removed from the edge.

3.2.2 Take Rule. The theory TakeRule defines theorems that establish whether, given a digraph, X can take read right to Z or take(read, X, Z, graph) as explained in List 3.8.

List 3.8: Take Rule

```
%Part 1:        SUBJECT ONLY: Rule defined 3 node graph
TakeRule [ Vertex: Type+]: THEORY
BEGIN
%Part 2:
Importing Definitions[Vertex]
Importing Graph_Init[Vertex]
Importing Add_Edge[Vertex]
Importing TG_Lemma_Init_Take[Vertex]

%Part 3: does edge (X,Y) have a take right
t_edge_in?: bool = FORALL(db:E_DB):
    edge?(InitGraph)(X,Y) and member(take,(ADD(db)(X,Y)))
```

```
%Part 4:  does edge(Y,Z) have a read right
r_edge_in?: bool = FORALL(db:E_DB):
    edge?(InitGraph)(Y,Z) and member(read,(ADD(db)(Y,Z)))

%Part 5: does edge (Y,X) have a take right
t_edge_in_for_tg_L?: bool = FORALL(db:E_DB):
    edge?(InitGraphL)(Y,X) and member(take,(ADDL(db)(Y,X)))

%Part 6: does edge(Y,Z) have a read right
r_edge_in_for_tg_L?: bool = FORALL(db:E_DB):
    edge?(InitGraphL)(Y,Z) and member(read,(ADDL(db)(Y,Z)))

%Part 7: adds the new edge into the graph
edge_taken: bool = FORALL(db:E_DB):
    edge?(AddEdge(InitGraph, X, Z))(X,Z) and
    member(read,(AddEdgeRight(ADD(db), X, Z, read)(X,Z)))

%Part 8: adds the new edge into the graph for the T-G Lemma
edge_taken_L: bool = FORALL(db:E_DB):
    edge?(AddEdge(InitGraphL, X, Z))(X,Z) and
    member(read,(AddEdgeRight(ADDL(db), X, Z, read)(X,Z)))

%THEOREMS:
%Part 9:    %Original Take-Grant rule(TAKE, X, Z, read)
Take_Edge: THEOREM
    (t_edge_in?  and r_edge_in?) iff edge_taken

%Part 10:    %used for the Take-Grant Lemma (Take X, Z, read)
Take_Edge_L: THEOREM
    (t_edge_in_for_tg_L? and r_edge_in_for_tg_L?) iff edge_taken_L

%Part 11:
END TakeRule
```

In Part3 *t_edge_in*? returns true or false depending on whether edge (X, Y) is in the digraph and has the take right. **edge?(InitGraph)(X, Y)** calls *InitGraph* which initializes and returns the graph which the *edge*? function from the digraphs library uses to determine if edge (X, Y) is in the digraph. Function **member(take, (ADD(db)(X, Y)))** calls *ADD* which initializes the rights the edges have, while *member* determines if the take right is a member of the edge (X, Y).

The *r_edge_in*? function in Part 4 is the same as Part 3 except it checks for the read right in edge (Y, Z).

The function *t_edge_in_for_tg_L*? in Part 5 returns true or false depending on if the edge (Y, X) is in the digraph and has the take right. **edge?(InitGraphL)(Y, X)** calls *InitGraphL* declaration to initialize the graph then *edge*? from the digraphs library determines if the edge (Y, X) is in the digraph. The **member(take, (ADDL(db)(Y, X)))** calls the *ADDL* declaration which initializes what rights the edges have, then *member* determines if take belongs to the edge (Y, X).

Part 6 *r_edge_in_for_tg_L*? is the same as Part 5 except it checks for the read right in edge (Y, Z).

In *edge_taken*, Part 7, returns true if the edge (X, Z) is in the digraph and that edge has the read right. The function: **edge?(AddEdge(InitGraph, X, Z))(X, Z)** initializes the digraph adds edge (X, Z). The function **member(read,(AddEdgeRight(ADD(db), X, Z, read)(X, Z)))** initializes rights to the current edges in the digraph then calls *AddEdgeRight* to add a right to the new edge. The *member* command checks to make sure it was added.

The function edge_taken_L, Part 8, is the same as edge_taken, Part 7, but uses the lemma graph.

Part 9 and 10 are the theorems for the Take rule.

The first theorem, Part 9 *Take_Edge*, proves the original Take-Grant Take rule. The theorem says if and only if (for the digraph in question) there is a take edge from X to Y and a read edge from Y to Z can the edge be taken. In this case edge (X, Y)'s

right is take and edge(Y, Z) has read. To simplify the theorem using the labels of the boolean functions already defined it is written **(t_edge_in? and r_edge_in?) iff edge_taken**. Figure 3.1 shows how the theorem transforms the original graph.

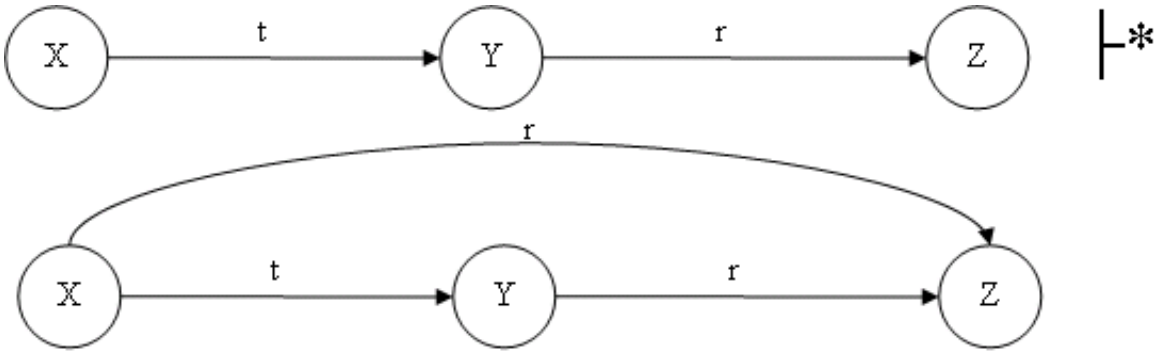

Figure 3.1: Results of theorem Take_Edge

The second theorem *Take_Edge_L* in Part 10 is used to prove the lemma to the Take-Grant Take rule. The theorem says that if and only if, for the chosen digraph, there is a take edge from Y to X and a read edge from Y to Z in the digraph can the edge be taken. To simplify the theorem using the labels of the boolean functions already defined is written as **(t_edge_in_for_tg_L? and r_edge_in_for_tg_L?) iff edge_taken_L** and shown in Figure 3.2.

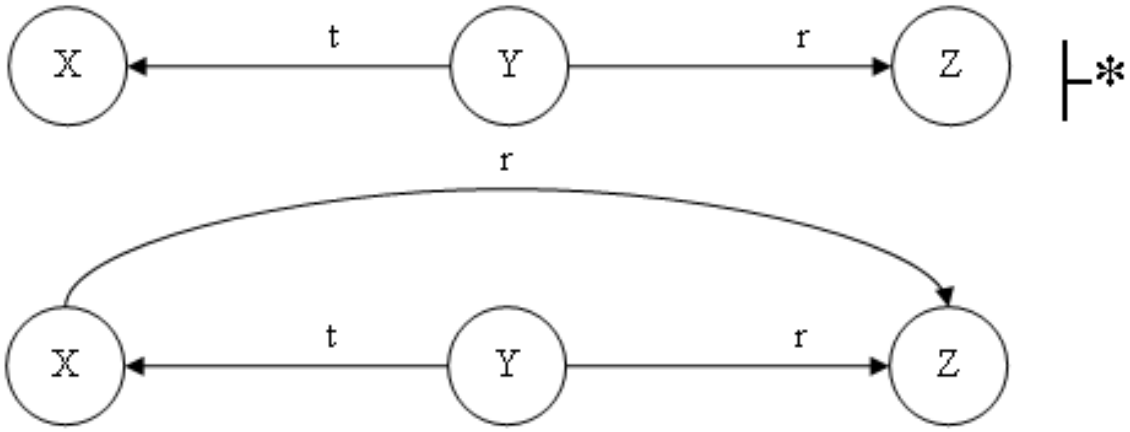

Figure 3.2: Results of theorem Take_Edge_L

3.2.3 Grant Rule. The GrantRule theory, List 3.9, is true if in the digraph specified X is granted read to Z. In this example, Y grants read to Z to X.

List 3.9: Grant Rule

```
%Part 1:
GrantRule [Vertex:Type+]: THEORY

%SUBJECT ONLY: Rule defined 3 node graph
BEGIN

%Part 2:
Importing Definitions[Vertex]
Importing Grant_Graph_Init[Vertex]
Importing Add_Edge[Vertex]
Importing TG_Lemma_Init_Grant[Vertex]

%Part 3: does edge (Y,X) have a grant right
g_edge_in?: bool = FORALL(db:E_DB):
    edge?(InitGraph)(Y,X) and member(grant,(ADD(db)(Y,X)))

%Part 4: does edge(Y,Z) have a read right
r_edge_in?: bool = FORALL(db:E_DB):
    edge?(InitGraph)(Y,Z) and member(read,(ADD(db)(Y,Z)))

%Part 5: does edge (X,Y) have a grant right
g_edge_in_for_tg_L: bool = FORALL(db:E_DB):
    edge?(InitGraphL)(X,Y) and member(grant,(ADDL(db)(X,Y)))


%Part 6: does edge(Y,Z) have a read right
r_edge_in_for_tg_L: bool = FORALL(db:E_DB):
    edge?(InitGraphL)(Y,Z) and  member(read,(ADDL(db)(Y,Z)))
```

```
%Part 7: adds the new edge into the graph
edge_granted: bool = FORALL(db:E_DB):
    edge?(AddEdge(InitGraph, X, Z))(X,Z) and
    member(read,(AddEdgeRight(ADD(db), X, Z, read)(X,Z)))

%Part 8:  adds the new edge into the graph
edge_granted_L: bool = FORALL(db:E_DB):
    edge?(AddEdge(InitGraph, X, Z))(X,Z) and
    member(read,(AddEdgeRight(ADDL(db), X, Z, read)(X,Z)))

%THEOREMS:
%Part 9: Original Take-Grant rule(Grant X, Z, read)
Grant_Edge: THEOREM
    (g_edge_in? and r_edge_in?) iff edge_granted

%Part 10: Used in the Take-Grant Lemma (Grant X, Z, read)
Grant_Edge_L: THEOREM
    (g_edge_in_for_tg_L and r_edge_in_for_tg_L)
    iff edge_granted_L

%Part 11:
END GrantRule
```

The function *g_edge_in*? in Part 3 returns true only if the edge (Y,X) is in the digraph and has the grant right. **edge?(InitGraph)(Y, X)** calls *InitGraph* to initialize the graph, then uses edge? function from the digraphs library determines if the edge (Y, X) is in the digraph. The **member(grant, (ADD(db)(Y, X)))** calls the *ADD* declaration which initializes what rights the edges have, then *member* from the sets library determine if grant belongs to the edge (Y, X).

Part 4 *r_edge_in*? is the same as Part 3 except it checks for the read right in edge (Y, Z).

In Part 5, *g_edge_in_for_tg_L*?, returns true or false depending on whether the edge (X, Y) is in the digraph and has the grant right. **edge?(InitGraphL)(X, Y)**

calls *InitGraphL* which initializes and returns the graph which the *edge*? function from the digraphs library uses to determine if edge (X, Y) is in the digraph. Function **member(grant, (ADDL(db)(X, Y)))** calls *ADDL* which initializes the rights edges have, then the *member* function determines if the grant right is a member of the edge (X, Y).

The function $r\_edge\_in\_for\_tg\_L$? in Part 6 is the same as Part 5 except it checks for the read right in edge (Y,Z).

The *edge_granted* function in Part 7 is true when the edge (X, Z) is in the digraph and has the read right. The function **edge?(AddEdge(InitGraph, X, Z))(X, Z)** initializes the digraph then adds edge (X, Z) to it. The function **member(read,(AddEdgeRight(ADD(db), X, Z, read)(X, Z)))** initializes the rights to the current edges in the digraph and calls *AddEdgeRight* to add a right to the new edge. The *member* command checks to make sure it was added.

In Part 8 *edge_granted_L* is the same as *edge_granted* but uses the lemma graph.

Part 9 and 10 are the theorems for the Grant rule.

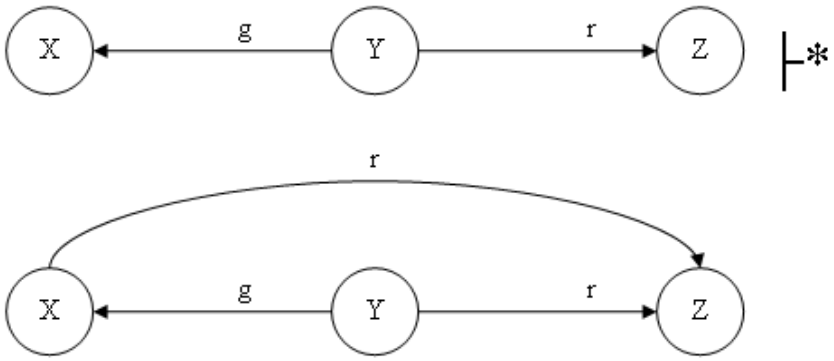

Figure 3.3: Results of theorem Grant_Edge

The first theorem in Part 9, *Grant_Edge*, proves the original Take-Grant Grant rule. The theorem says that if and only if (for the current digraph) there is a grant edge and a read edge from X to Z in the digraph can the edge be granted. To

simplify the theorem using the labels of the boolean functions already defined it is written **(g_edge_in? and r_edge_in?) iff edge_granted**. Figure 3.3 shows how the theorem transforms the original graph.

The second theorem *Grant_Edge_L* in Part 10 proves a lemma for the Take-Grant Grant rule. The theorem says that if and only if (for the initialized digraph) there is a grant edge and a read edge from X to Z in the digraph can the edge be granted. To simplify the theorem using the labels of the boolean functions already defined it is written **(g_edge_in_for_tg_L and r_edge_in_for_tg_L) iff edge_granted_L** and is shown in Figure 3.4

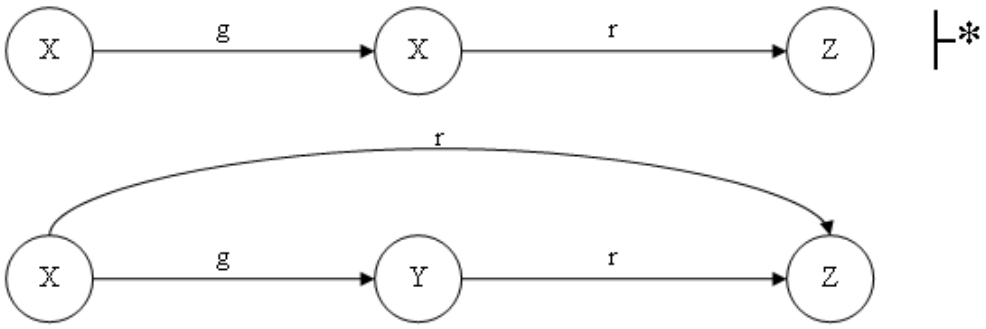

Figure 3.4: Results of theorem Grant_Edge_L

3.2.4 Create Rule. The CreateRule theory, List 3.10 returns true or false if the vertex specified X can create vertex A and give the new edge (X, A) all the rights. Written as create(allrights, X, A, graph), where all rights are read, write, take, and grant.

List 3.10: Create Rule

```
%Part 1:                %SUBJECT ONLY: Rule defined
CreateRule    [ Vertex: TYPE+ ]:THEORY
BEGIN
```

```
%Part 2:
Importing Definitions[Vertex]
Importing Graph_Init[Vertex]
Importing Add_Edge[Vertex]

%Part 3: checks to see if vert exists that is doing the creation
v_exists?: bool = member(X, vert(InitGraph))

%Part 4: checks to see if vert to be created exists
new_v_not_exists?: bool = FORALL(A:Vertex|A/=X and A/=Y and A/=Z):
    not member(A, vert(InitGraph))

%Part 5: new vert is created
v_created: bool = FORALL(A:Vertex|A/=X and A/=Y and A/=Z):
    member(A, vert(AddVert(InitGraph, A)))

%Part 6: new edge is created
e_created: bool =FORALL(A:Vertex|A/=X and A/=Y and A/=Z):
    edge?(AddEdge(AddVert(InitGraph, A),X, A))(X,A)

%Part 7: new edge has allrights added to it
e_r_created:bool =  FORALL(db:E_DB):
    member(all_rights,(AddEdgeAllRights(ADD(db), X, A)(X,A)))

%Part 8: (Create,X,A, allrights)  - allrights=(r,w,t,g)
Create_Vert_Right: THEOREM
    ((v_exists? and new_v_not_exists?) iff v_created)
    implies (e_created and e_r_created)

%Part 9: (Create,X,A, allrights)  - allrights=(r,w,t,g)
Created: THEOREM
    (v_exists? and new_v_not_exists?) implies
    (v_created and e_created and  e_r_created)
%Part 10:
END CreateRule
```

The function *v_exists*? in Part 3 returns true or false based on whether the vertex doing the creating exists in the digraph. Function **member(X, vert(InitGraph))** checks that X is in the graph.

In Part 4, *new_v_not_exists*? returns true or false depending on if the vertex to be created is already a member of the digraph. This function states that all vertices are not equal with a FORALL declaration: **FORALL(A:Vertex—A/=X and A/=Y and A/=Z)**. The digraph is initialized and through the negation of the *member* function A it is determined A is not part of the digraph. This function is **not member(A, vert(InitGraph))**.

Part 5 *v_created* returns true if the new vertex was successfully added to the digraph. In function **member(A, vert(AddVert(InitGraph, A)))** the graph is initialized, then the $AddVert$ function is called with that digraph and the vertex to add A. Once the vertex is created, the *member* function checks to see if the vertex was added.

The *e_created* function in Part 6 is true if the new edge was successfully created from the creating vertex to the new vertex. The graph is initialized with $InitGraph$ in the function **edge?(AddEdge(AddVert(InitGraph, A),X, A))(X, A)** then $AddVert$ and $AddEdge$ is called. Finally *edge*? function confirms that the edge was added to the digraph.

In Part 7 *e_r_created* returns true or false depending on if the new edge (X, A) contains all the rights: read, write, take, and grant. The function **member(all_rights,(AddEdgeAllRights(ADD(db), X, A)(X, A)))** first initializes the edge database, then $AddEdgeAllRights$ is called with the new edge to add rights to. Finally *member* checks to make sure the correct rights were added to (X, A)'s edge database.

Part 8 and 9 are the theorems for the Create rule. Even though the two theorems are different, the change to the graph produced by each theorem is fundamentally the same and shown in Figure

Create_Vert_Right in Part 8 is the first theorem to prove the Take-Grant Create rule. It states that if and only if the creating vertex exists and the vertex to be created does not then the vertex can be added to the graph, which implies that the edge was created and all the rights were added to the new edge database. To simplify the theorem, the boolean functions already defined were used: **((v_exists? and new_v_not_exists?) iff v_created) implies (e_created and e_r_created)**.

3.5.

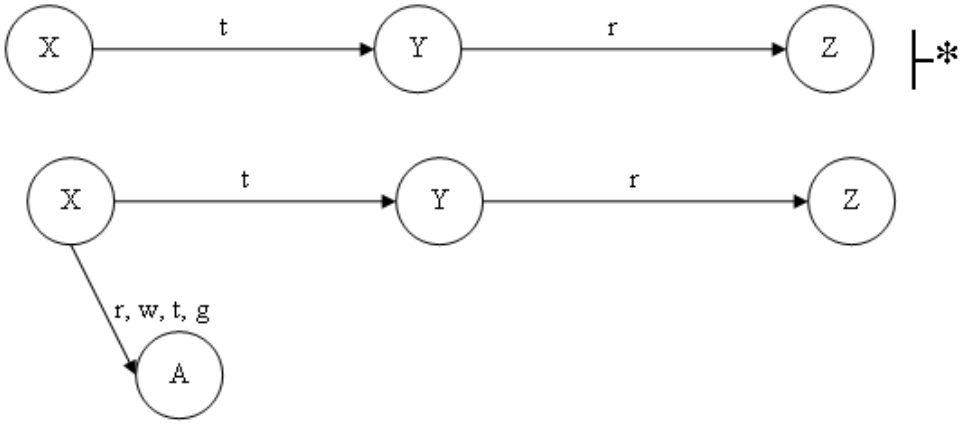

Figure 3.5: Create Theorem

Part 9 is the *Created* theorem, which uses a different approach to prove the Take-Grant Create rule. It states that if the creation vertex exists and the vertex to be created does not then the vertex can be added to the graph, the new edge was created connecting the two vertices, and all the rights were added to the new edge edge database. The theorem is written **(v_exists? and new_v_not_exists?) implies (v_created and e_created and e_r_created)**, since the boolean functions were already defined.

3.2.5 Remove Rule. The RemoveRule theory in List 3.11 is true for a specified digraph if the read right for edge (Y, Z) is removed. If there are no more rights for that edge, the edge itself is removed. The command is written remove(Y, Z, read).

List 3.11: Remove Rule

```
%Part 1:
RemoveRule   [ Vertex: TYPE+ ] : THEORY
%SUBJECT ONLY: Rule defined
BEGIN

%Part 2:
 Importing Definitions[Vertex]
 Importing Graph_Init[Vertex]
 Importing RemoveEdge[Vertex]

%Part 3: does right exist to be removed
r_exists?: bool = FORALL(db:E_DB):
    member(read,(ADD(db)(Y,Z)))

%Part 4: does the edge exist to remove right
e_exists?: bool = edge?(InitGraph)(Y,Z)

%Part 5: right is removed
r_removed: bool = FORALL(db:E_DB):
    not member(read,(RemoveEdgeRight(ADD(db), (Y,Z), read)(Y,Z)))

%Part 6: edge is removed if it has no rights
e_removed: bool = FORALL(db:E_DB):
    empty?((RemoveEdgeRight(ADD(db), (Y,Z), read)(Y,Z))) iff
    (not edge?(RemoveEdge(InitGraph,(Y,Z)))(Y,Z))


 %THEOREM:
%Part 7: Rule (Remove, Y,Z,read)
Remove_Right: THEOREM
    (e_exists? and r_exists?) implies (r_removed and e_removed)

%Part 8:
END RemoveRule
```

Part 2 states the theories that Remove Rule uses is: Definitions, Take_Graph_Init, and RemoveEdge. Although any graph initialization scheme could be used, RemoveRule uses Take_Graph_Init.

In Part 3, the *r_exists* function returns true or false depending on the existence of the right to be removed in the edge database. **member(read,(ADD(db)(Y, Z)))** initially calls *ADD* to add all the rights to all the edges in the digraph, then using the *member* function makes sure that read is a right on the (Y, Z) edge.

The function in Part 4, *e_exists*, returns true or false based on whether the edge the right belongs to exists in the graph. **edge?(InitGraph)(Y, Z)** initializes the digraph then using the *edge*? function checks to see if (Y, Z) exists as an edge in the digraph.

Part 5 *r_removed* returns true if the edge to be removed was successfully removed. *r_removed* uses **not member(read, (RemoveEdgeRight(ADD(db), (Y, Z), read)(Y, Z)))** which first calls *ADD* to add all the rights to the edges in the digraph. Then it calls *RemoveEdgeRight* which removes the right in question. Using not in front of the *member* function causes it to return true if the edge is no longer a member in (Y, Z) right set.

The *e_removed* function, Part 6, returns true or false depending on whether the edge is removed if and only if it has no rights. **empty?((RemoveEdgeRight(ADD(db), (Y, Z), read)(Y, Z)))** first calls *ADD* to add all the rights to the edges. Then calls *RemoveEdgeRight* removes the right. Using the *empty*? function the edge right set checks to see if its empty if and only if that is true is the second part of the specification considered. **(not edge?(RemoveEdge(InitGraph,(Y, Z)))(Y, Z))** will initialize the graph then call *RemoveEdge* to delete the required edge (Y,Z) and then check by negating the *edge*? function to make sure it is no longer a member in the graph.

The theorem in Part 7 proves the Remove rule: *Remove_Right*. The theorem says that if the right to be removed exists and the edge it exists on is in the digraph,

this implies the right can be removed and the edge as well if the edge contains no other rights. The theorem is simplified by using the boolean functions already defined and looks like **(e_exists? and r_exists?) implies (r_removed and e_removed)**. Figure 3.6 shows how the theorem transforms the graph.

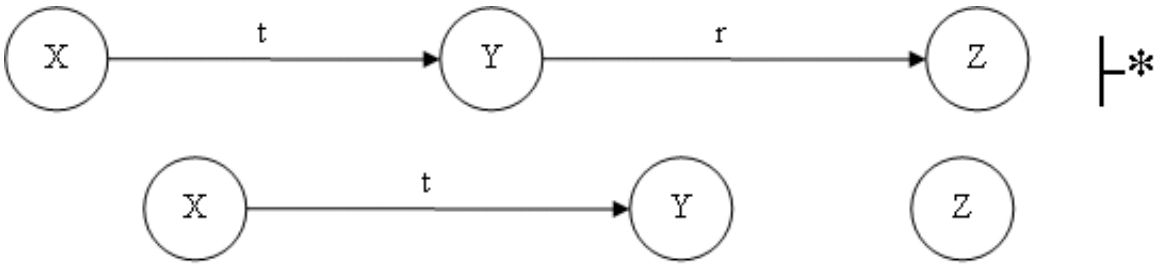

Figure 3.6: Remove Theorem

3.3 Take-Grant Model Two - Error Checking

Model two allows an arbitrary digraph to be specified. Recursion is used to move through a case statement which allows more error checking.

3.3.1 Common Imported Theories. The theories that are imported for the rules are listed here. Only if the theories changed between models are they explained below. The Take_Graph_Init theory initializes the digraph and the edge database for both the Take, Create, and Remove rules. The only difference is which definition file is imported: tgDefinitions is used for the Take rule, while cDefinitions is used by Create and rDefinitions is imported for the Remove rule. Grant_Graph_Init along with Add_Edge are exactly the same theories used for Model 1.

tgDefinitions Theory

The tgDefinitions theory contains all the definitions used for the Take and Grant rule in List 3.12. In the following, only those aspects of the PVS specification that are new are discussed.

List 3.12: tgDefinitions Theory

```
%Part 1:
tgDefinitions [Vertex: TYPE+]: THEORY
  BEGIN

%Part 2:
Importing digraphs@digraphs[Vertex]
Importing digraphs@digraph_ops[Vertex]
Importing digraphs@digraph_deg[Vertex]

%Part 3: %declares TYPE of right
Rights: TYPE = {read, write, take, grant}

%Part 4: declares a function: given an edge returns the rights ...
   that belong to it
E_DB: TYPE =function[edgetype[Vertex]->set[Rights]]

%Part 5: Vertices that can be used in the graph
X,Y,Z,A: Vertex

%Part 6:
nil:Vertex

%Part 7: Program counter Type
 PCT: TYPE = {L0,L1,L2,L3,L4,L5,L6,L7,LTRUE,LFALSE,LEND}

%Part 8: attributes for each vertex
node : TYPE = [#
            dest: Vertex, % destination
            source: Vertex, % source edge starts from
            e1: edgetype, % an outgoing edge
            ie: finite_set[edgetype], % all incoming edges
            oe: finite_set[edgetype] % all outgoing edges
            #]
```

```
%Part 9: node database function given a vertex returns its ...
   attributes
n_db: TYPE = function[Vertex -> node]

%Part 10: labelled graph attributes
 L_Graph: TYPE =     [#
            g1: digraph[Vertex],
            db: E_DB,
            PC: PCT,
            Node: n_db,
            taken: bool,
            addright: Rights,
            granted:bool
            #]

%Part 11:
null: edgetype = (nil,nil)

%Part 12: AXIOM: which states the vertices aren't equal
 not_eq_ax: axiom X/=Y and Y/=Z and Z/=X

%Part 13:
END tgDefinitions
```

Part 2 imports the digraph, digraph_ops, and digraph_deg theories, instantiated for the Vertex type. All of those theories were developed by NASA Langley.

Part 6 declares a nil vertex, which is used to mark dead-ends in the functions

In Part 7 the values used for the program counter are declared. These values move the recursion through the case statement.

Part 8 declares a node type, which contains all the attributes for the node. The attributes that each node has are: *dest* which contains the destination node or the node that is used to progress down the digraph, *source*, which is either the same as the node or is the node that was previously used in the state, *ie* which is a set of

all incoming edges to the vertex, and *oe*, which is a set of all outgoing edges to the vertex.

The node database, Part 9, declares as a function which given a vertex returns the node attributes.

Part 10 is a type of L_Graph (labelled graph) which holds all the variables for the current state. This is the main type. The variables that L_Graph has are: $g1$, which is the changed digraph; *db*, which is the edge database; PC, which is the program counter; $Node$, which contains the node database; *taken*, which is used for the Take rule and will be true if the edge can be taken; *addright*, an addright variable; and *granted*, which is used for the Grant rule and will be true if the edge can be granted.

Part 11 declares a null edgetype which is designated to be (nil,nil) an empty edgetype

cDefinitions Theory

The cDefinitions theory in List 3.13 contains all the definitions used for the Create rule.

List 3.13: cDefinitions Theory

```
%Part 1:
cDefinitions  [ Vertex: Type+ ] : THEORY

BEGIN

%Part 2:
Importing digraphs@digraphs[Vertex]
Importing digraphs@digraph_ops[Vertex]
Importing digraphs@digraph_deg[Vertex]

%Part 3: declares TYPE of right works
Rights: TYPE = {read, write, take, grant}
```

```
%Part 4: declares a function given an edge returns the rights that...
    belong to it
E_DB: TYPE =function[edgetype[Vertex]->set[Rights]]

%Part 5: Vertex 's that can be used in the graph
X,Y,Z,A,B: Vertex

%Part 6:
nil: Vertex

%Part 7:
 PCT: TYPE = {L0,L1, L2, L3, L4,L5,LTRUE,LFALSE,LEND}

%Part 8:
L_Graph: TYPE =     [#
            g1: digraph[Vertex],
            db: E_DB,
            PC: PCT,
            created: bool
            #]

%Part 9:
null: edgetype = (nil,nil)

%Part 10: AXIOM : which states the vertices aren't equal
not_eq_ax: axiom X/=Y and Y/=Z and Z/=X and X/=A and A/=Y and A/=Z

%Part 11:
END cDefinitions
```

In Part 7, the PC values are not the same as tgDefinitions, so that theory cannot be used, because there are more values than case statements you get a Type Correctness Condition (TCC) is generated when proving.

In Part 8, the only different between tgDefinitions is the variable *created* which will be true if a new vert/edge can be created.

rDefinitions Theory

In List 3.14 the rDefinitions theory defines all the definitions used for the Remove rule.

List 3.14: rDefinitions Theory

```
%Part 1:
rDefinitions  [ Vertex: Type+ ]: THEORY
  BEGIN

%Part 2:
Importing digraphs@digraphs[Vertex]
Importing digraphs@digraph_ops[Vertex]
Importing digraphs@digraph_deg[Vertex]

%Part 3: declares TYPE of right works
Rights: TYPE = {read, write, take, grant}

%Part 4:
E_DB: TYPE =function[edgetype[Vertex]->set[Rights]]

%Part 5:
X,Y,Z,A,B: Vertex

%Part 6:
nil: Vertex

%Part 7:
PCT: TYPE = {L0,L1,L2,L3,L4,LTRUE,LFALSE,LEND}
```

```
%Part 8:
 L_Graph: TYPE =      [#
             g1: digraph[Vertex],
             db: E_DB,
             PC: PCT,
             removed: bool
             #]

%Part 9: null: edgetype = (nil,nil)

%Part 10:
not_eq_ax: axiom X/=Y and Y/=Z and Z/=X

%Part 11:
END rDefinitions
```

Part 7 declares the values used for the program counter. These values move the recursion through the case statement.

Part 8 declares a type L_Graph which the only difference from cDefinitions is that *removed* is used instead of *create*. *Removed* is true if the right to be removed was successfully removed.

Node_ops Theory

The Node_ops theory, List 3.15, defines all the operations on nodes. This theory is used only for the Take and Grant rules.

List 3.15: Node_ops Theory

```
%Part 1:
Node_ops [Vertex: TYPE+]: THEORY

  BEGIN

%Part 2:
Importing tgDefinitions[Vertex]
```

```
%Part 3:
set_edge(x:Vertex,a:edgetype, c:L_Graph,oes: finite_set[edgetype],
ies: finite_set[edgetype]): L_Graph = c
    with [‘Node(x)‘e1:=a, ‘Node(x)‘dest:= a‘2,‘Node(x)‘source:=x, ...
        ‘Node(x)‘oe:=oes, ‘Node(x)‘ie:=ies ]

%Part 4:    %sets node attributes
get_edge(c: L_Graph, x: Vertex): L_Graph = if x=X
    then set_edge(x, (X,Y), c,outgoing_edges(x, c‘g1),...
       incoming_edges(x, c‘g1))
else  (if x=Y then set_edge(x,(Y,Z),c,outgoing_edges(x,
c‘g1),incoming_edges(x, c‘g1))
    else    set_edge(x,(Z,Z),c,outgoing_edges(x, c‘g1),...
       incoming_edges(x, c‘g1))
endif) endif

%Part 5:    %update node dest and source
Node_update(Node: n_db, x: Vertex, e1: Vertex, e2: Vertex):n_db =
Node with [(x)‘source:= e1,(x)‘dest:= e2]

%Part 6:    %for each vertex the node attributes are set
set_Node_e1(c: L_Graph, v: finite_set[Vertex]): recursive L_Graph=
    if empty?(v)then   c
    else let a = choose(v) in
        set_Node_e1(get_edge(c, a), remove(a,v))
endif measure  card(v)

%Part 7:
END Node_ops
```

In Part 3, the *set_edge* function sets the nodes $e1$, *dest*, *source*, *oe* and *ie* variables. The $e1$ variable contains an edge for the node, *source* is set to itself, *dest*

is set to the edges second value, *oe* contains the outgoing edges and ie contains the nodes incoming edges.

The *get_edge* function in Part 4 calls the *set_edge* function. Different edges are passed to *set_edge* depending on which vertex was used to call *get_edge*.

Part 5 the *Node_update* function updates the source and destination for the node that calls it.

In Part 6 the *set_Node_e*1 function is a recursive call to go through all the vertices in the graph and set their node attributes.

RemoveEdge Theory

The Model 2 RemoveEdge theory, List 3.16, defines all remove edge and right functions. it is basically the same as the RemoveEdge specification for Model 1, List 3.7, except for the Remove_Edge function.

List 3.16: RemoveEdge Theory

```
%Part 1:
RemoveEdge [ Vertex: Type+]: THEORY

BEGIN

%Part 2:
Importing rDefinitions[Vertex]

%Part 3: For REMOVE - removes edges
RemoveEdge(g1: digraph, e:edgetype):  digraph[Vertex] =  g1 with
[edges:= remove(e,edges(g1))]

%Part 4: Removes rights to an edge
RemoveEdgeRight(db: E_DB, e:edgetype, r: Rights):E_DB = db with
[(e`1,e`2):= remove(r, db(e))]
```

```
%Part 5: Main function call to delete edge
Remove_Edge(c:L_Graph,e:edgetype):L_Graph =
    if edge?(c`g1)(e)
    then c with [ g1:= RemoveEdge((c`g1),e)]
    else c
    endif

%Part 6:
END RemoveEdge
```

The *Remove_Edge* function in Part 5 is the function that decides if the edge needs to be removed. The functions parameters are in L_Graph and an edgetype variable. If the edge exists in the current digraph, it is removed, else the digraph is returned with no change.

3.3.2 Take Rule. The take_rule theory in List 3.17 returns true or false if in the digraph specified, X can take read rights to Z from Y.

List 3.17: Take Rule

```
%Part 1:
take_rule [Vertex: TYPE+]: THEORY

%SUBJECT ONLY
%Is used strictly for the 3 node query, which is how the rule is ...
   defined.

BEGIN

%Part 2:
Importing tgDefinitions[Vertex]
Importing Graph_Init[Vertex]
Importing Node_ops[Vertex]
Importing Add_Edge[Vertex]
```

```
%Part 3:    %loop actions
%Part 3.1:
L0(c: L_Graph, x:Vertex, y:Vertex): L_Graph = c with
    [g1:=InitGraph, db:= ADD(c`db)]

%Part 3.2:
L3(c: L_Graph, x: Vertex, y:Vertex):L_Graph =
    set_Node_e1(c,vert(c`g1))

%Part 3.3:
%see if the current edge has take for a right if it does update ...
   the destination and source
Take_out_edge?(c: L_Graph, x: Vertex, n: n_db): L_Graph =
    if  member(take, c`db(c`Node(x)`e1))
    then c with [Node:= Node_update(c`Node, x, c`Node(x)`e1`1, c`...
       Node(x)`e1`2)]
    else c endif

%Part 3.4
L5(c: L_Graph, x: Vertex, r:Rights):L_Graph = c with
[Node:= Node_update(c`Node, x, c`Node(x)`e1`1, c`Node(x)`e1`2),
addright:= r]

%Part 3.5    %adds new edge to graph, adds new edge right to ...
   edge_db
L6(c: L_Graph, x: Vertex, y: Vertex, r: Rights):L_Graph = c with [
    g1:=AddEdge(c`g1, x, y),
    db:= AddEdgeRight(c`db, x, y, r)]

%Part 3.6
LTRUE(c:L_Graph): L_Graph = c with [taken:= true]

%Part 3.7
LFALSE(c:L_Graph): L_Graph = c with [taken:= false]
```

```
%Part 4: %Loop
sw(c: L_Graph, r: Rights, x: Vertex,  y: Vertex): L_Graph =
    Cases c‘PC of

%Part 4.1:            %initializes graph and E_DB
    L0:L0(c,x,y) with [PC:=L1],

%Part 4.2: % checks if first node is in graph else quit
    L1: c with [PC:= if not member(x, vert(c‘g1)) then LFALSE
            % checks if second node is in graph else quit
        else (if not member(y, vert(c‘g1)) then LFALSE
            % checks if needed edge exists else need to obtain ...
               edge
        else(if not member((x,y),outgoing_edges(x, c‘g1)) then L2
            % checks if needed edge has right needed else go ...
               obtain edge with correct right
        else (if member(r,(c‘db(x,y))) then LFALSE else L2 endif)
        endif)  endif)  endif],

%Part 4.3: %checks if there are at least 3 verts in graph else ...
   quit
    L2: c with [PC:= if (card[Vertex](vert(c‘g1))=3)
            then L3
            else LFALSE endif],

%Part 4.4: %initializes the vertices attributes
    L3: L3(c, x, y) with [PC:= L4],

%Part 4.5:
    L4: Take_out_edge?(c, x, c‘Node)  with [PC:=L5],

%Part 4.6: %updates X’s destination node’s destination, sets the ...
   right to add to edge
    L5: L5(c, Y, r) with [PC:=L6],
```

```
%Part 4.7: %adds edge to graph and adds right edge to database
    L6: L6(c, x, y, r) with [PC:=L7],

%Part 4.8: %checks to see if right exists for added edge
    L7: c with [PC:= if member(r,(c`db(x,y))) then LTRUE else ...
       LFALSE endif],

%Part 4.9:
     LTRUE: LTRUE(c) with [PC:= LEND],

%Part 4.10:
    LFALSE:LFALSE(c) with [PC:= LEND],

%Part 4.11:
    LEND: c

    Endcases

%Part 5:
takerule(num: nat, r: Rights, x: Vertex, y: Vertex, initial:
L_Graph ): Recursive L_Graph = if num=0 then
    initial with [PC:=L0]
else
    sw(takerule(num-1,r,x,y,initial),r,x,y)
endif measure num

%Part 6:
Take_Rule_Taken: THEOREM FORALL (initial: L_Graph): FORALL(num:
nat | takerule(num, read, X, Z, initial)`PC=LEND): takerule(num,
read, X, Z, initial)`taken=true

%Part 7:
END take_rule
```

Part 2 states that this theory is going to use the tgDefinitions, Graph_Init, Node_ops, and Add_Edge theories.

Part 3 are the loop actions taken when for the PC functions are taken

Part 3.1 L0 initializes the graph $g1$ and the rights for the edge db.

Part 3.2 L3 calls set_Node_e1 function which will set the node attributes for each vertex in the graph.

Part 3.3 $Take\_out\_edge$ verifies if the take right is in the current selected edge, if it is, then the source and destination node are updated for that vertex else the L_Graph is returned unchanged.

Part 3.4 L5 updates X's destination node's (Y) destination and source.

Part 3.5 L6 adds the new edge to the graph and the edge with right to the edge database.

Part 3.6 LTRUE returns L_Graph with taken set to true.

Part 3.7 LFALSE returns L_Graph with taken set to false.

Part 4 is the recursive loop built into a case statement. The recursion moves as the PC is changed through each iteration. The case statement is called sw which takes a L_Graph, and two vertices, and a right. The vertex that is going to take the given right from the second vertex.

Part 4.1 L0 calls loop action L0 and sets PC to L1.

Part 4.2 L1 checks to make sure both vertices are in the digraph and that the edge does not exist with the correct right. If it does exist then LFALSE if the edge does not exist with correct right then L2.

Part 4.3 L2 checks to see if there are three vertices in the graph. If there are 3 vertices then the PC is set to L3 otherwise it is set to LFALSE.

Part 4.4 L3 calls the loop action L3 to initialize the vertices attributes.

Part 4.5 L4 loop action Take_out_edge is called and PC=L5.

Part 4.6 L5 updates the node's destination destination node, in this case Y and then goes to L6.

Part 4.7 L6 adds the new edge to the graph then calls L7.

Part 4.8 L7 checks to make sure the edge created has the required right. If it does, it goes to LTRUE else LFALSE.

The LTRUE PC counter in Part 4.9 goes to LEND. This indicates that *taken* was indeed successful.

Part 4.10, the PC counter LFALSE goes to LEND. This indicates that *taken* failed.

The PC counter LEND, Part 4.11, signals the end of the program counter and returns a L_Graph.

In Part 5, *takerule* is the recursive call to go through the program counter case statements found in the *sw* function. If $num = zero$ then the initial graph starts with the PC = L0 else the case statement *sw* is called with *takerule* as one of the parameters, which invokes the recursion.

$Take\_Rule\_Taken$ in Part 6 is the theorem for Take, which says that for all the initial L_Graph, for all *num* (arbitrary number used for recursion), when the PC=LEND $taken = true$.

3.3.3 Grant Rule. The grant_rule theory is designed to return true or false in List 3.18 if for the digraph specified Y grants read rights to Z to X.

List 3.18: Grant Rule

```
%Part 1: SUBJECT ONLY. is used for rule definition: 3 node query
 grant_rule [Vertex : TYPE+] : THEORY
BEGIN

%Part 2:
Importing tgDefinitions[Vertex]
Importing Grant_Graph_Init[Vertex]
```

```
Importing Node_ops[Vertex]
Importing Add_Edge[Vertex]

%Part 3:    %loop actions
%Part 3.1:
L0(c: L_Graph, x:Vertex , y:Vertex ): L_Graph = c with
    [g1:=InitGraph, db:= ADD(c‘db)]

%Part 3.2:
L3(c: L_Graph, x: Vertex , y:Vertex ):L_Graph =
    set_Node_e1(c, vert(c‘g1))

%Part 3.3:
%see if the current edge has grant  for a right if it does update ...
   the dest and source
Grant_in_edge?(c: L_Graph, x: Vertex , n: n_db): L_Graph =
    if  member(take, c‘db(c‘Node(x)‘e1))
    then c with [Node:= Node_update(c‘Node, x, c‘Node(x)‘e1‘1, c‘...
       Node(x)‘e1‘2)]
    else c endif

%Part 3.4:
L5(c: L_Graph, x: Vertex , r:Rights):L_Graph = c with [Node:=
Node_update(c‘Node, x, c‘Node(x)‘e1‘1, c‘Node(x)‘e1‘2), addright:=
r]

%Part 3.5: %adds new edge to graph, adds new edge right to ...
   right_db
L6(c: L_Graph, x: Vertex , y: Vertex , r: Rights):L_Graph = c with
[    g1:=AddEdge(c‘g1, x, y),
    db:= AddEdgeRight(c‘db, x, y, r)]

%Part 3.6:
LTRUE(c:L_Graph): L_Graph = c with [granted:= true]
```

```
%Part 3.7:
LFALSE(c:L_Graph): L_Graph = c with [granted:= false]

%Part 4:  Loop
sw(c: L_Graph, r: Rights, x: Vertex ,  y: Vertex ): L_Graph =

    Cases c‘PC of
%Part 4.1:  %initializes graph and E_DB
    L0:L0(c,x,y) with [PC:=L1],

%Part 4.2:  % checks if first node is in graph else quit
    L1: c with [PC:= if not member(x, vert(c‘g1)) then LFALSE
            % checks if second node is in graph else quit
        else (if not member(y, vert(c‘g1)) then LFALSE
            % checks if needed edge exists else obtain new edge
        else(if not member((x,y),outgoing_edges(x, c‘g1)) then L2
            % checks if correct right exists on edge
        else (if member(r,(c‘db(x,y))) then LFALSE else L2 endif)
        endif)  endif)  endif],

%Part 4.3:  %checks there are at least 3 verts in graph else quit
    L2: c with [PC:= if (card[Vertex ](vert(c‘g1))=3)
            then L3
            else LFALSE endif],
%Part 4.4:
    L3: L3(c, x, y) with [PC:= L4],

%Part 4.5:
    L4: Grant_in_edge?(c, x, c‘Node)  with [PC:=L5],

%Part 4.6: updates X’s dest node’s dest, sets the right to add to ...
   edge
    L5: L5(c, Y, r) with [PC:=L6],
```

```
%Part 4.7: %adds edge to graph and adds right edge to database
    L6: L6(c, x, y, r) with [PC:=L7],

%Part 4.8:
    L7: c with [PC:= if member(r,(c`db(x,y))) then LTRUE else ...
       LFALSE endif],

%Part 4.9:
    LTRUE: LTRUE(c) with [PC:= LEND],

%Part 4.10:
    LFALSE:LFALSE(c) with [PC:= LEND],

%Part 4.11:
    LEND: c

    Endcases

%Part 5:
grantrule(num: nat, r: Rights, x: Vertex , y: Vertex , initial:
L_Graph ): Recursive L_Graph = if num=0 then
    initial with [PC:=L0]
else
    sw(grantrule(num-1,r,x,y,initial),r,x,y)
endif measure num

%Part 6:
Grant_Rule_Granted: THEOREM FORALL (initial: L_Graph): FORALL(num:
nat | grantrule(num, read, X, Z, initial)`PC=LEND): grantrule(num,
read, X, Z, initial)`granted=true

%Part 7:
END grant_rule
```

Part 2 states that this theory uses the tgDefinitions, Grant_Graph_Init, Node_ops, and Add_Edge theories.

Part 3 are the loop actions taken for the PC functions.

Part 3.1 L0 initializes the graph $g1$ and the rights for the edge db.

Part 3.2 L3 calls set_Node_e1 function which will set the node attributes for each vertex in the graph.

Part 3.3 *Grant_in_edge* verifies the grant right is in the current selected edge. If it is, then the source and destination nodes are updated for that vertex. Otherwise the L_Graph is returned unchanged.

Part 3.4 L5 updates the X's destination node's (Y) destination and source.

Part 3.5 L6 adds the new edge to the graph and the edge with right to the edge database.

Part 3.6 LTRUE returns L_Graph with taken set to true

Part 3.7 LFALSE returns L_Graph with taken set to false

Part 4 The recursive loop is built into a case statement, the recursion moves as the PC is changed through each iteration. The case statement is called *sw* which takes a L_Graph, and two vertices, and a right. The vertex that is going to take the given right from the second vertex.

Part 4.1 L0 calls loop action L0 and sets PC to L1

Part 4.2 L1 checks to make sure both vertices are in the digraph and that the edge does not exist with the correct right. If it does exist then PC is set to LFALSE otherwise it is set with L2.

Part 4.3 L2 checks if there are three vertices in the graph, if there are then go to L3 else quit by setting PC to LFALSE.

Part 4.4 L3 calls the loop action L3 to initialize the vertices attributes.

Part 4.5 L4 loop action Grant_in_edge is called and PC=L5.

Part 4.6 L5 updates the node's destination destination node, in this case Y and then goes to L6.

Part 4.7 L6 adds the new edge to the graph then calls L7.

Part 4.8 L7 checks to make sure the edge created has the required right. If it does, it goes to LTRUE theorem is successful otherwise theorem failed and PC goes to LFALSE.

Part 4.9 LTRUE goes to LEND. This indicates that granted was indeed successful.

Part 4.10 LFALSE goes to LEND. This indicates that granted failed.

Part 4.11 LEND is then end of the program counter and returns a L_Graph.

Part 5 *grantrule* is the recursive call to go through the program counter case statements found in the *sw* function. If $num = zero$ then the initial graph starts with the PC = L0 else the case statement *sw* is called with *grantrule* as one of the parameters, which invokes the recursion.

Part 6 *Grant_Rule_Granted* is the theorem for Take, which says for all the initial L_Graph, for all some *num* (arbitrary number used for recursion) that when the PC=LEND $granted = true$.

3.3.4 Create Rule. The CreateRule theory is specified in List 3.19 and returns true or false if for the digraph specified, X can create vertex A and give the new edge (X, A) all the rights written as create(X, A, allrights), where all rights are read, write, take, and grant.

List 3.19: Create Rule

```
%Part 1:
create_rule    [ Vertex: TYPE+ ]: THEORY
%SUBJECT ONLY

BEGIN
```

```
%Part 2:
Importing cDefinitions[Vertex]
Importing Add_Edge[Vertex]
Importing Graph_Init[Vertex]

%Part3:  loop actions
%Part3.1:
L0(c: L_Graph): L_Graph = c with [g1:=InitGraph, db:= ADD(c‘db) ]

%Part 3.2: creates the new VERT for graph
CREATE_VERT(c: L_Graph, x: Vertex,y:Vertex):L_Graph = c with
[g1:=AddVert(c‘g1, y)]

%Part 3.3: creates the new edge for the new vert created.
CREATE_EDGE(c: L_Graph, x: Vertex, y: Vertex):L_Graph = c with [
    g1:=AddEdge(c‘g1, x, y),
    db:= AddEdgeAllRights(c‘db, x, y)]

%Part 3.4:
LTRUE(c:L_Graph): L_Graph = c with [created:= true]

%Part 3.5:
LFALSE(c:L_Graph): L_Graph = c with [created:= false]

%Part 4:    %Loop
sw(c: L_Graph, x: Vertex,  y: Vertex): L_Graph =
    Cases c‘PC of

%Part 4.1:
        L0:L0(c) with [PC:=L1],

%Part 4.2:  % checks if first node is in graph else quit
        L1: c with [PC:=if not member(x, vert(g1(c)))
            then LFALSE else L2 endif],
```

```
%Part 4.3:  %creates the new node
        L2: CREATE_VERT(c,x,y) with [PC:=L3],

%Part 4.4:
        L3: c with [PC:=if  member(y, vert(g1(c))) then L4
                else LFALSE endif],

%Part 4.5:  %creates the new edge to the new node.
        L4:CREATE_EDGE(c, x, y) with [PC:=L5],

%Part 4.6:
        L5: c with [PC:= if member((x,y),edges(g1(c))) then
                LTRUE else LFALSE endif],

%Part 4.7:
        LTRUE:LTRUE(c) with [ PC:= LEND],

%Part 4.8:
        LFALSE:LFALSE(c) with [ PC:= LEND],

%Part 4.9:
        LEND: c

    Endcases

%Part 5:
can_create(num: nat, x: Vertex, y: Vertex, initial:
L_Graph ): Recursive L_Graph = if num=0 then
    initial with [PC:=L0]
else
    sw(can_create(num-1,x,y,initial),x,y)
endif measure num
```

```
%Part 6:
Create: THEOREM FORALL (initial: L_Graph):
FORALL(num: nat | can_create(num, X, A, initial)`PC=LEND):
can_create(num, X, A, initial)`created = true

%Part 7:
END create_rule
```

Part 2 states that this theory is going to use the cDefinitions, Graph_Init, and Add_Edge theories.

Part 3 are the loop actions taken when for the PC functions.

Part 3.1 L0 initializes the graph $g1$ and also the rights for the edge db.

Part 3.2 $CREATE\_VERT$ returns L_Graph with the new vertex added to $g1$.

Part 3.3 $CREATE\_EDGE$ returns the L_Graph with the new edge added to $g1$ and the rights for the new edge added to the edge database.

Part 3.4 LTRUE returns L_Graph with created set to true

Part 3.5 LFALSE returns L_Graph with created set to false

Part 4 is the recursive loop built into a case statement. The recursion moves as the PC is changed through each iteration. The case statement is called sw which takes a L_Graph, and two vertices. The vertex that is going to create the new vertex and the new vertex to be created.

Part 4.1 L0 calls loop action L0 and sets PC to L1.

Part 4.2 L1 if the vert that is doing the creating is not a member of $g1$ then the PC is set to LFALSE else it is set to L2.

Part 4.3 L2 loop action $CREATE\_VERT$ is called and PC is set to L3.

Part 4.4 L3 If the new vert has been added successfully then PC equals L4 otherwise PC is set to LFALSE because the vertex was not created.

Part 4.5 L4 loop action $CREATE\_EDGE$ is called and PC equals L5.

Part 4.6 L5 if the new edge has been successfully added then PC equals LTRUE else set PC to LFALSE.

Part 4.7 LTRUE goes to LEND. This states that Create was indeed successful.

Part 4.8 LFALSE goes to LEND. This states that Create failed.

Part 4.9 LEND is then end of the program counter and returns a L_Graph.

Part 5 *can_create* is the recursive call to go through the program counter case statements found in the *sw* function. If $num = zero$ then the initial graph starts with the PC = L0 else the case statement *sw* is called with *can_create* as one of the parameters, which invokes the recursion.

Part 6 *Can_Create* is the theorem for Create, which says that for all the initial L_Graph, for all *num* (arbitrary number used for recursion), when the PC=LEND $created = true$.

3.3.5 Remove Rule. In List 3.20 the RemoveRule theory returns true if, for the digraph specified, the read right for edge (Y, Z) is removed. If there are no more rights for that edge, the edge is also removed.

List 3.20: Remove Rule

```
%Part 1:   SUBJECT ONLY
remove_rule   [ Vertex: TYPE+ ] : THEORY

BEGIN

%Part 2:
Importing rDefinitions[Vertex]
Importing Graph_Init[Vertex]
Importing RemoveEdge[Vertex]

%Part 3:   loop actions
%Part 3.1:
L0(c: L_Graph): L_Graph = c with [g1:=InitGraph, db:= ADD(c`db) ]
```

```
%Part 3.2:
Remove_Right(c:L_Graph,e:edgetype,r:Rights): L_Graph=
        %checks to see if the edge has any rights
    if empty?(c‘db(e))
        %no rights delete edge
    Then Remove_Edge(c,e)
        %has rights see if it has right we need
    Else (if member(r,c‘db(e))
        %has right now remove it
    Then c with [db:=RemoveEdgeRight((c‘db), e, r), removed:=true]
    else c with [removed:=false]
    endif)    endif

%Part 3.3:
LTRUE(c: L_Graph): L_Graph = c with [removed:= true]

%Part 3.4:
LFALSE(c: L_Graph): L_Graph = c with [removed:= false]

%Part 4:     %Loop
sw(c: L_Graph, r: Rights, x: Vertex,  y: Vertex): L_Graph =
    Cases c‘PC of

%Part 4.1:  %initializes graph and E_DB
        L0:L0(c) with [PC:= L1],

%Part 4.2:  % checks if first node is in graph else quit
        L1: c with [PC:= if not member(x, vert(c‘g1))
            then LFALSE
                % checks if second node is in graph else quit
            else (if not member(y, vert(c‘g1)) then LFALSE
                % checks if needed edge exists else need to obtain...
                    edge
            else L2 endif)  endif],
```

```
%Part 4.3: %checks if the edge even exists in graph.
        L2: c with [PC:= if edge?(c`g1)(x,y) then  L3
            else LFALSE endif],

%Part 4.4:  %removes right or  actual edge depending on graph.
        L3: Remove_Right(c, (x,y),r) with [PC:= L4],

%Part 4.5:
        L4: c with [PC:=
            if empty?(c`db((x,y))) then L3
            else LEND endif ],

%Part 4.6:
        LTRUE: LTRUE(c) with [PC:= LEND],

%Part 4.7:
        LFALSE: LFALSE(c) with [PC:= LEND],

%Part 4.8:
        LEND: c
Endcases

%Part 5:
can_remove(num: nat, r: Rights, x: Vertex, y: Vertex,
initial: L_Graph ): Recursive L_Graph = if num=0 then
    initial with [PC:=L0]
else
    sw(can_remove(num-1,r,x,y,initial),r,x,y)
endif measure num

%Part 6:
%(Remove, take, X, Y)
Can_Remove: THEOREM FORALL (initial: L_Graph):
FORALL(num: nat | can_remove(num, take, X, Y, initial)`PC=LEND):
can_remove(num, take, X, Y, initial)`removed = true
```

```
%Part 7:
END remove_rule
```

Part 2 states that this theory is going to use the cDefinitions, Graph_Init, and RemoveEdge theories.

Part 3 are the loop actions taken for the PC functions.

Part 3.1 L0 initializes the graph $g1$ and also the rights for the edge db.

Part 3.2 *Remove_Right* returns a L_Graph with possibly $g1$, db, and removed changed. First it checks to see if the edge has any rights. If it does not, it calls *Remove_Edge* to delete the edge else it checks to see if the right to be deleted is a part of the right to that edge. If the right exists then *RemoveEdgeRight* is called, and the right is removed and removed equals true, else removed equals false.

Part 3.3 LTRUE returns L_Graph with removed set to true.

Part 3.4 LFALSE returns L_Graph with removed set to false.

Part 4 is the recursive loop built into a case statement. The recursion moves as the PC is changed through each iteration. The case statement is called *sw* which takes a L_Graph, the right to be removed, and two vertices. The two vertices are used to get the edge to delete the right from.

Part 4.1 L0 calls loop action L0 and sets PC to L1.

Part 4.2 L1 if both the vertices passed in as parameters exist as a member of $g1$ then the PC is set to L2 otherwise it is set to LFALSE.

Part 4.3 L2 If the edge to delete the right from exists in $g1$ then PC equals L3, if not then PC is set to LFALSE because the edge does not exist.

Part 4.4 L3 loop action *Remove_Right* is called and PC equals L4.

Part 4.5 L4 if the edge that the right was removed from has no more rights then PC equals L3 else PC equals LEND.

Part 4.6 L5 if the new edge has been successfully added then PC=LTRUE else LFALSE.

Part 4.7 LTRUE goes to LEND, which means Remove was indeed successful.

Part 4.8 LFALSE goes to LEND. This means Remove failed.

Part 4.9 LEND is the end of the program counter and returns a L_Graph.

Part 5 *can_remove* is the recursive call that goes through the program counter case statements found in the *sw* function. If *num* = zero then the initial graph starts with the PC = L0 else the case statement *sw* is called with *can_remove* as one of the parameters, which invokes the recursion.

Part 6 *Can_Remove* is the theorem for Remove, which says that for all the initial L_Graph, for all *num* that when PC=LEND removed = true.

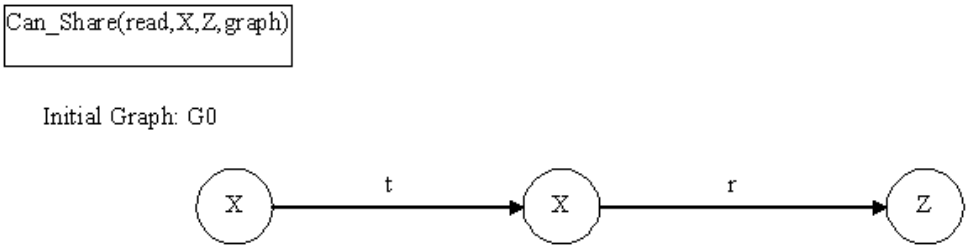

Figure 3.7: Example 1 of Can_Share Graph G_0

3.4 Can_Share Algorithm

The Can_Share is one of the four main predicates for the Take-Grant Model. It formally defines the notion of transferring authority [Bis95]. Thus, it is important to implement in PVS. The Can_Share algorithm, List 3.21 uses Figure 3.7 as a reference even though the algorithm is designed to operate with any number of vertices in the digraph.

List 3.21: Can_Share Algorithm

```
Can_Share(read,X,Z,graph)
***Can have a t* or a t*g* path

L0: Initialize the graph and the rights for the edges. If X is not
in digraph then goto LFALSE Else if Z is not in graph then goto
LFALSE
    Else if edge (X,Z)  does  not  exist then goto L1
    Else  if edge(X,Z) contain the read right  then goto LTRUE
        Else goto L1

L1: if there is  at least three vertices in the graph then goto L2
    Else goto LFALSE

L2:  if Z does nothave incoming edges then goto  LFALSE else goto
L3

L3: if X does nothave any edges then goto LFALSE else  goto L4


L4: if X  has an edge that  either has a take or a grant right
then save the other vertex that makes up that edge  in a variable
into variable TWO and goto L5 Else goto LFALSE

L5: if an edge exists from TWO to Z with the read right
then goto L6
Else  If an edge exists with a take right: save the corresponding
edge vertex into THREE and goto  L8
else if an edge exists with a grant right: save the corresponding ...
   edge vertex into THREE
goto L9 else goto L4

L6: Add edge(X,Z) with right read to digraph and edge database
goto L7
```

```
L7: If edge(X,Z) exists in the graph with read right goto LTRUE

L8: add edge(X,THREE) with take right to digraph and edge database
goto L4

L9: add edge(X,THREE) with grant right to digraph and edge
database goto L4

LTRUE Can_Share(read,X,Z,graph) is true

LFALSE - Can_Share(read,X,Z,graph) is false
```

L0 initializes the graph and the edge database then checks to make sure the two vertices to share rights exist in the digraph and that an edge does not already exist between them with the correct right. If an edge already exists with the right wanted then *Can_Share* is true, otherwise it continues on with the algorithm.

L1 checks to make sure that there is at least three vertices in the digraph otherwise the right cannot be shared and *Can_Share* is false. If it is true go to L2.

L2 checks to see if Z has incoming edges. If Z has no incoming edges, there is no way X can get shared read rights to it. Therefore, *Can_Share*, is false. Otherwise go to L3.

L3 checks to see if X has any edges. If it does, go to L4. Otherwise *Can_Share* is false. Even though *Can_Share* uses a digraph for the take and grant rights direction is not a concern because of the Take-Grant lemmas.

L4 find an edge leaving X that has either a take or a grant right. In our example edge (X, Y) has a take so save vertex Y into variable TWO and goto L5. If there are no edges, the *Can_Share* is false.

L5 finds an edge that exists from the TWO variable to either Z with the correct right and goes to L6. Or it looks for an edge with a take and goes to L8 or an edge

with a grant and goes to L9. If there are no edges with the correct rights and go to L4 to check if there are anymore edges from X.

L6 adds the edge (X, Z) with read right to the graph and edge database and goto L7.

L7 double checks to make sure that the edge was successfully added to the digraph then goes to LTRUE else LFALSE.

L8 and L9 are divided so the proper chain of witnesses can be established. For an example, consider Figure 3.8 which shows Can_Share from (X, Z), but only by going through vertex A so L8 and L9 are for this situation.

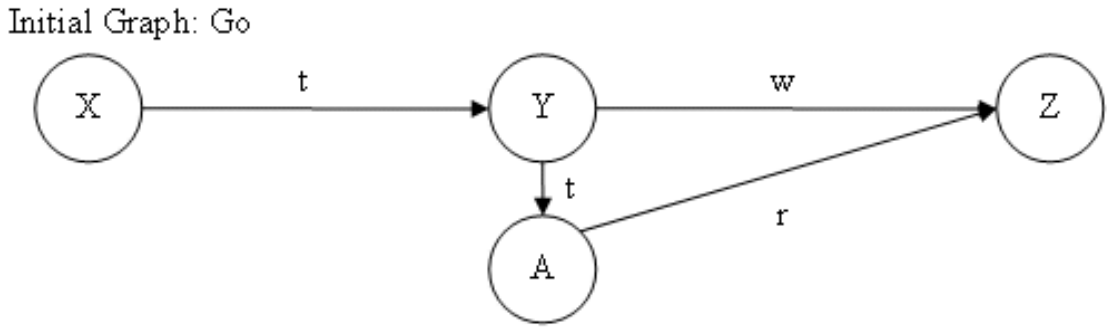

Figure 3.8: Example 2 Can_Share Graph G_0

L8 adds edge(X,THREE) with a take right to the digraph and edge database. Goto L4 to continue trying to share read rights to Z to X.

L9 adds edge(X,THREE) with a grant right to the digraph and edge database. Goto L4 to continue trying to share read rights to Z to X.

LTRUE means that *Can_Share* is true therefore an edge from (X,Z) with read right can be added or X can obtain the read right.

LFALSE means that *Can_Share* is false and an edge from (X,Z) with read right can NOT be added.

This algorithm operates on the assumption that no matter how many vertices are in the digraph, only three are examined at a time: hence the variable TWO and THREE keep track of where the algorithm is in the digraph.

3.5 Contributions to PVS Digraph Library

The Labeled_digraph theory is an extension of the digraph theory. Labeled_digraph allows labels to be assigned directly to the edge and keeps them together like a record. There is really only two differences between digraph and labeled_digraph: digraph has only one parameter of type T and labeled_digraph has two parameters T and U, where U is assumed to be an enumerated type.

```
1. edgetype: TYPE = pair[T]

2. predigraph: TYPE = [# vert  : finite_set[T],
                        edges: finite_set[edgetype] #]

3. directed_graph: TYPE = {g: predigraph | FORALL e: edges(g)(e)
    IMPLIES LET (x,y) = e IN vert(g)(x) AND vert(g)(y)}

4. digraph: TYPE = directed_graph

5. simple_digraph: TYPE = {g: digraph | FORALL e: edges(g)(e)
    IMPLIES LET (x,y) = e IN x /= y}

6. sd_graph: TYPE = simple_digraph

7. edge?(G)(x,y): bool = edges(G)((x,y))

8. edges_vert: LEMMA in?(x,e) AND edges(G)(e)
    IMPLIES (EXISTS y: vert(G)(y) AND in?(y,e))

9. other_vert: LEMMA in?(v,e) AND edges(G)(e)
    IMPLIES (EXISTS (u: T): vert(G)(u) AND e = (u, v) OR e = (v,u))
```

Figure 3.9: Partial Digraph Theory.

In digraph, the edgetype is made up of a pair of T. In a labeled_digraph edgetype is a record made up of two fields: edge, which is a pair of T and label, which is a finite set of U. Pairs and Finite Sets are functions in existing PVS theories: pair and finite_set. All regular digraph functions and lemmas become labeled_digraph functions. In Figure 3.9 and Figure 3.10 selected definitions and functions are shown

to highlight the difference between digraph theory and labelled_digraph. For both figures, corresponding numbers are used in the discussion. Line 1 shows the difference in how edgetype is defined. In digraph it is a simple pair, and in labeled_digraph it is a record holding two fields: edge and label. For both figures lines 2, 4, and 6 involve only a name change. For lines 3, 5, 7, 8, 9 the difference in the theories is in digraph *edge* is used, however in labelled_digraph $e'edge$ is used. Because labelled_digraph declares edgetype as a record, access to edge has to be done by using the field name.

Lines 3 and 7 are different in labeled_digraphs because of record accession. To access the edge, first an edgetype has to be accessed, then the individual field has to be accessed, which is done through $'1$ or $'2$. For example $e'edge'1$ accesses the first field of edge which is a field in edgetype.

```
1. edgetype: TYPE = [# edge: pair[T], label: finite_set[U] #]

2. pre_labeled_digraph: TYPE = [# vert: finite_set[T],
                                  edges: finite_set[edgetype] #]

3. directed_labeled_graph: TYPE =
    {g: pre_labeled_digraph | FORALL e: edges(g)(e)
    IMPLIES vert(g)(e`edge`1) AND vert(g)(e`edge`2)}

4. labeled_digraph: TYPE = directed_labeled_graph

5. simple_labeled_digraph: TYPE =
    {g: labeled_digraph | FORALL e: edges(g)(e)
    IMPLIES e`edge`1 /= e`edge`2}

6. sld_graph: TYPE = simple_labeled_digraph

7. edge?(G)(x, y): bool =
    EXISTS (e: edgetype): edges(G)(e) AND e`edge`1 = x AND e`edge`2 = y

8. edges_vert:
    LEMMA in?(x, e`edge) AND edges(G)(e)
    IMPLIES (EXISTS y: vert(G)(y) AND in?(y, e`edge))

9. other_vert:
    LEMMA in?(v, e`edge) AND edges(G)(e)
    IMPLIES
    (EXISTS (u: T): vert(G)(u) AND e`edge = (u, v) OR e`edge = (v, u))
```

Figure 3.10: Partial Labelled Digraph Theory

This contribution, which was to incorporate labels into the digraph structure, was submitted to NASA Langley for inclusion in their PVS libraries, however it was not used for the thesis because it was developed late in the effort and the structure for the Take-Grant was already firmly established.

3.6 Summary

The two specification models are described. Chapter 4 explains the proof of the theorems. Since the Can_Share algorithm was discussed in this chapter, the actual code will be discussed in the next.

IV. Proving the Take-Grant Model in PVS

4.1 Introduction

PVS has an integrated proof checker to verify specifications. "The primary emphasis in the PVS proof checker is on supporting the construction of readable proofs [SORSC99]." As such, much attention was given to simplifying the process of developing, debugging, maintaining, and presenting proofs. To assist in developing proofs, powerful proof commands carry out propositional, equality, and arithmetic reasoning. Due to the integration between the typechecker and the proof checker, PVS supports an expressive specification language, while also providing a powerful theorem proving capability [SORSC99]. "The PVS typechecker analyzes theories for semantic consistency and adds semantic information to the internal representation built by the parser [OSRSC99b]."

Theorem proving may be required to establish the type-consistency of a PVS specification, therefore the typechecker uses the deductive power of the proof checker to discharge any proof obligations, termed type correctness conditions (TCCs), generated. TCCs may arise when a term is typechecked against an expected predicate subtype or as subgoals during proof checking, when the typechecker is invoked to check user-supplied expressions and quantifier instantiations [SORSC99]. The TCCs do not have to be proved immediately, but until they are proved, the theory that generated them is not considered to be typechecked and any theorems proved are considered to be "proved-incomplete" [OSRSC99b]. The proof representation and PVS prover commands are discussed in the following subsections.

4.1.1 PVS Proof Representation. PVS proofs follow standard proof theory when displayed on the screen. The following overview of the proof representation is taken from the PVS Prover Guide [SORSC99][1] :

> PVS has a sequent-style proof representation, which allows the effects of the prover commands to be understood. A proof tree is maintained,

[1]For a more indepth understanding of the prover reference PVS Prover Guide, [SORSC99]

with the goal being that a proof tree is constructed with all the leaves being true, thus it is "complete". Each leave/node is a proof goal in the proof tree from which, by means of a proof step, spring its children. Each proof goal is a sequent consisting of a sequence of formulas called antecedents and a sequence of formulas called consequents.

In PVS, such a sequent is displayed in List 4.1 where the Ai and Bj are PVS formulas collectively referred to as sequent formulas: the Ai are the antecedents and the Bj are the consequents; the row of dashes serves to separate the antecedents from the consequents. In text sequents are written: $A1, A2, A3, ... \vdash B1, B2, B3...$

List 4.1: PVS Sequent Example [SORSC99]

```
      {-1] A1
      {-2} A2
      [-3] A3
          .
          .
          .
----------------
      {1] B1
      [2] B2
      {3} B3
          .
          .
          .
```

The sequence of antecedents or consequents (but not both) may be empty. The intuitive interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents, i.e., $(A1 \bigwedge A2 \bigwedge A3...) \supset (B1 \bigvee B2 \bigvee B3...)$. The proof tree starts with a root node of the form $\vdash A$, where A is the theorem to be proved. PVS proof steps build a proof tree by adding subtrees to leaf nodes as directed by the proof commands. A sequent is true if any antecedent is the same as any consequent, if any antecedent is false, or if any consequent is true. Other sequents can also be recognized as true, using more powerful inferences that will be described later. Once a sequent is recognized as true, that branch of the proof tree is terminated. The goal is to build a proof tree whose branches have all been terminated in this way.

Attention is always focused on some sequent that is a leaf node in the current proof tree–this sequent is displayed by the PVS prover while awaiting the user's command. The numbers in brackets, e.g., [-3], and braces, e.g., 3, before each formula in the displayed sequent are used to name the corresponding formulas. The formula numbers in square brackets indicate formulas that are unchanged in a subgoal from the parent goal whereas the numbers in braces serve to highlight those formulas that are either new or different from those of the parent sequent.

The proof structure makes it easy for the user to understand the effects the proof command described in the next section have on the proof.

4.1.2 PVS Commands. The commands used to prove the two specifications are **grind**, **skosimp**, **induct**, **lemma**, and **expand**. Their definitions are described below.

The *grind* strategy has the syntax "**(grind)**" which rewrites formulas and repeatedly simplifies them. "This is a catch-all strategy that is frequently used to automatically complete a proof branch or to apply the obvious simplifications till they no longer apply [SORSC99]." Grind is the short command that applies in order: $install - rewrites$, *bddsimp*, *assert*, $replace*$, and *reduce*. The command $install - rewrites$ installs given theories and rewrite rules along with all the relevant definitions in the given subgoal. *Bddsimp*, which stands for binary decision diagram simplification, applies propositional simplification and *assert* simplifies using the decision procedures to carry out the first level of simplification [SORSC99]. The $replace*$ command carries out all the equality replacements, and the $*$ means the command will be iteratively applied. The command reduce is the "main workhorse" of the *grind* command and repeatedly simplifies through the application of the *bash* and $replace*$ command [SORSC99]. The *bash* command is another shortcut command executing in order *assert*, *bddsimp*, *inst*?, $skolem - typepred$, $flatten$, and $lift-if$ commands [SORSC99]. The command *inst*? instantiates existential strength quantifiers, while $skolem - typepred$ skolemizes with type constraints, and $lift - if$ rule lifts the left-innermost contiguous IF or CASES branching structure to the top level [SORSC99].

Skosimp command which has the syntax **(skosimp)** is the short version of **(skolem)(flatten)**. *Skolem* replaces universal quantifiers with constants and $flatten$ performs disjunctive simplification [SORSC99].

The *induct* command syntax is **(induct var)**, which automatically applies an induction scheme. The variable name **var** must be quantified at the outermost level

of the consequent formula. As long as the bound variable is of a type, such as nat, the induction scheme is selected automatically. Most induction schemes are found in the PVS prelude library [OS03].

The *lemma* command, **(lemma name)**, incorporates lemma **name** in to the proof. Additional subgoals may be generated with the addition of the lemma rule [SORSC99].

The *expand* command syntax: **(expand name)** expands and simplifies all instances of the **name** [SORSC99].

This remaining sections explain how to prove the TCCs, the theorems for both specifications, as well as discusses the Can_Share algorithm.

```
% Subtype TCC generated (at line 8, column 57) for
    % g1 WITH [edges := remove(e, edges(g1))]
    % expected type  digraph[Vertex]
  % unfinished
RemoveEdge_TCC1: OBLIGATION
  FORALL (g1: digraph[Vertex], e: edgetype[Vertex]):
    FORALL (e_1: edgetype[Vertex]):
      remove[edgetype[Vertex]](e, g1`edges)(e_1) IMPLIES
       g1`vert(e_1`1) AND g1`vert(e_1`2);
```

Figure 4.1: RemoveEdge_TCC1

4.2 Proving the Type Correctness Conditions

Even though two separate specifications were developed, the common imported theories used in each were very similar, therefore both have the same TCCs to be proved.

The theory RemoveEdge produces the RemoveEdge_TCC1 proof obligation, Figure 4.1, which states that for all $g1$ digraphs and e_1 edgetypes, if the remove function removes e from $g1$'s edges set then both of the vertices are in the graph. This TCC is easily discharged with the grind command. Figure 4.2 shows what the

```
  |-------
{1}   FORALL (g1: digraph[Vertex], e: edgetype[Vertex]):
        FORALL (e_1: edgetype[Vertex]):
          remove[edgetype[Vertex]](e, edges(g1))(e_1) IMPLIES
           g1`vert(e_1`1) AND g1`vert(e_1`2)

Rule? (grind) /= rewrites e /= e_1
  to NOT (e = e_1)
member rewrites member(e_1, edges(g1))
  to edges(g1)(e_1)
remove rewrites remove[edgetype[Vertex]](e, edges(g1))(e_1)
  to NOT (e = e_1) AND edges(g1)(e_1)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Figure 4.2: RemoveEdge_TCC1 Proof

RemoveEdge_TCC1 looks like in PVS prover. At the prompt "Rule?" the (*grind*) is entered which proves the TCC ending in Q.E.D..

The Add_Edge theory produces the proof obligation Add_Edge_TCC1 of Figure 4.3, and of AddVert_TCC1, Figure 4.4. The AddEdge TCC states that given two distinct vertices in the graph, any edges added to the graph is connecting vertices already in the graph. This TCC is proved through an application of the *grind* command.

```
FORALL (g1: digraph[Vertex], x, y: Vertex):
  g1`vert(x) AND g1`vert(y) IMPLIES
  (FORALL (e: edgetype[Vertex]):
    add[edgetype[Vertex]]((x, y), g1`edges)(e) IMPLIES
     g1`vert(e`1) AND g1`vert(e`2));
```

Figure 4.3: AddEdge_TCC1

Figure 4.4, the AddVert TCC, states that if a vertex is not already in the graph, then when it is added there is an edge that connects the vertex to itself. The *grind* command also discharges this TCC.

The Graph_Init and Grant_Graph_Init both produce the InitGraph_TCC1 because the initialization function is called InitGraph. However for each theory the TCC

```
% Subtype TCC generated (at line 26, column 77) for
    % g1 WITH [vert := add(x, vert(g1))]
    % expected type  digraph[Vertex]
  % unfinished
AddVert_TCC1: OBLIGATION
 FORALL (g1: digraph[Vertex], x: Vertex):
 NOT g1`vert(x) IMPLIES
 (FORALL (e: edgetype[Vertex]):
  g1`edges(e) IMPLIES
  add[Vertex](x, g1`vert)(e`1) AND add[Vertex](x, g1`vert)(e`2));
```

Figure 4.4: AddVert_TCC1

has to be solved because the function is different, due to edge order. The same goes for TG_Lemma_Init_Take and TG_Lemma_Init_Grant and the InitGraphL_TCC1.

In Figure 4.5 the obligation states it must be proven that for any edge added to the graph, each vertex is added first to $e'1$ and then added to $e'2$ so they show up in both pairs. Applying the *grind* command discharges this TCC .

```
FORALL (e: edgetype[Entity]):
  add[edgetype[Entity]]
      ((X[Entity], Y[Entity]),
       add[edgetype[Entity]]
           ((Y[Entity], Z[Entity]), emptyset[edgetype[Entity]]))
      (e)
   IMPLIES
   add[Entity]
       (X[Entity], add[Entity](Y[Entity], singleton[Entity](Z[...
          Entity])))(e`1)
    AND
    add[Entity]
        (X[Entity], add[Entity](Y[Entity], singleton[Entity](Z[...
           Entity])))
        (e`2);
```

Figure 4.5: InitGraph_TCC1

There are ten TCCs which are particular to the second specification two each from Node_ops, take, grant, CREATE and the Remove theory. Node_ops produces two TCCs for set_Node_e1 TCC1, Figure 4.6, and set_Node_e1 TCC1, Figure 4.7.

Figure 4.6, the first TCC, is to establish that v is a finite set. If it is not empty then it is non-empty and therefore has values in it. This can be discharged by applying

```
% Subtype TCC generated (at line 33, column 21) for  v
    % expected type  p: (nonempty?[Entity])
  % unchecked
set_Node_e1_TCC1: OBLIGATION
  FORALL (v: finite_set[Entity]): NOT empty?(v) IMPLIES nonempty?[...
     Entity](v);
```

Figure 4.6: set_Node_e1_TCC1

the *grind* command. The second TCC, Figure 4.7, validates that the not empty finite set v implies a value can be chosen out of the set and this decreases the cardinality of the set. This TCC is discharged through the prover commands *(grind)(rewrite "card_remove")(assert)*.

```
 FORALL (v: finite_set[Entity]):
   NOT empty?(v) IMPLIES
    (FORALL (a: (v)):
       a = choose(v) IMPLIES
        card[Entity](remove[Entity](a, v)) < card[Entity](v));
```

Figure 4.7: set_Node_e1_TCC2

Each of the main rules produce two TCC's for their theorems. The take_rule produces takerule_TCC1 and takerule_TCC2; grant_rule produces grantrule_TCC1 and grantrule_TCC1; CREATE produces can_create_TCC1 and can_create_TCC2, while Remove produces can_remove_TCC1 and can_remove_TCC2. However, the only difference between the TCC1s are the name, likewise with TCC2s. These TCCs establish that the recursion stops. In Figure 4.8 the obligation is to prove that for all *num*'s, *num* is not equal to zero implies that *num* minus 1 will be greater than or equal to zero.

```
% Subtype TCC generated (at line 95, column 13) for  num - 1
    % expected type  nat
  % unchecked
takerule_TCC1: OBLIGATION FORALL (num: nat): NOT num = 0 IMPLIES
num - 1 >= 0;
```

Figure 4.8: takerule_TCC1

The second TCC in Figure 4.9 is to prove that all *num*'s are not equal to zero and *num* minus one is less than *num*. Both of these TCCs are discharged with the grind command.

```
% Termination TCC generated (at line 95, column 4) for
    % takerule(num - 1, r, x, y, initial)
  % unchecked
takerule_TCC2: OBLIGATION FORALL (num: nat): NOT num = 0 IMPLIES
num - 1 < num;
```

Figure 4.9: takerule_TCC2

After all the TCCs are proved then each theorem can be "proved-complete" once it itself proves.

```
%Original Take-Grant rule(TAKE, X, Z, read)
Take\_Edge: THEOREM
    (t\_edge\_in?  and r\_edge\_in?) iff edge\_taken

%used for the Take-Grant Lemma (Take X, Z, read)
Take\_Edge\_L: THEOREM
    (t\_edge\_in\_for\_tg\_L? and r\_edge\_in\_for\_tg\_L?)
    iff edge\_taken\_L

%Original Take-Grant rule(Grant X, Z, read)
Grant\_Edge: THEOREM
    (g\_edge\_in? and r\_edge\_in?) iff edge\_granted

%Used in the Take-Grant Lemma (Grant X, Z, read)
Grant\_Edge\_L: THEOREM
    (g\_edge\_in\_for\_tg\_L and r\_edge\_in\_for\_tg\_L)
    iff edge\_granted\_L
```

Figure 4.10: Take and Grant Rule Theorems

4.3 Take-Grant Model One - No Error Checking - Proofs

The Model one specification does not incorporate automatic error checking. The theorems are straightforward and require knowledge of graph for the manual input and decision making.

Both of the theorems in TakeRule, along with the GrantRule, Figure 4.10, use the commands (*grind*) and *(lemma "not_eq_ax").* Depending on the order of the commands either *(grind)(lemma "not_eq_ax")(grind)* or (*lemma*"*not_eq_ax*")(*grind*) will work. The former was first used, because it was not apparent immediately the vertices would need to be proved not equal. In the first case (*assert*) can also be substitute for the last *grind*. The *lemma* command can be invoked first, then the *grind* command which takes slightly less time.

The two theorems in CreateRule and the one in RemoveRule, Figure 4.11, only use the (*grind*) rule. CreateRule would have used the lemma rule however A is declared in CreateRules functions to not be equal to the other vertices.

```
%(Create,X,A, allrights)  - allrights=(r,w,t,g)
Create\_Vert\_Right: THEOREM
    ((v\_exists? and new\_v\_not\_exists?) iff v\_created)
    implies (e\_created and e\_r\_created)

%(Create,X,A, allrights)  - allrights=(r,w,t,g)
Created: THEOREM
    (v\_exists? and new\_v\_not\_exists?) implies
    (v\_created and e\_created and  e\_r\_created)

%Rule (Remove, Y,Z,read)
Remove\_Right: THEOREM
(e\_exists? and r\_exists?) implies (r\_removed and e\_removed)
```

Figure 4.11: Create and Remove Theorems

4.4 Take-Grant Model Two - Error Checking - Proofs

This specification incorporates automatic error checking. The theorems use recursion to facilitate the error checking thus no prior knowledge of the graph is needed to test the theorems. However, problems were found in the Take and Grant rules, when attempting to use a variable to specify a value for a function. Either the variables caused a non-terminating recursion or the right proof commands could not be determined. Because the Take and Grant rule should only be run on a 3

vertex graph, which is how the basic rules are defined, hard coding the vertices is not a problem if the X,Y,Z graph never changes. The Create and Remove rules are straightforward and do not need prior knowledge of the graph to run. The theorems listed in Figure 4.12 are similar except for the variable used to prove true.

```
Take\_Rule\_Taken: THEOREM FORALL (initial: L\_Graph):
FORALL(num: nat | takerule(num, read, X, Z, initial)`PC=LEND):
takerule(num, read, X, Z, initial)`taken=true

Grant\_Rule\_Granted: THEOREM FORALL (initial: L\_Graph):
FORALL(num: nat | grantrule(num, read, X, Z, initial)`PC=LEND):
grantrule(num, read, X, Z, initial)`granted=true

Create: THEOREM FORALL (initial: L\_Graph):
FORALL(num: nat | can\_create(num, X, A, initial)`PC=LEND):
can\_create(num, X, A, initial)`created = true

Can\_Remove: THEOREM FORALL (initial: L\_Graph):
FORALL(num: nat | can\_remove(num, take, X, Y, initial)`PC=LEND):
can\_remove(num, take, X, Y, initial)`removed = true
```

Figure 4.12: Take, Grant, Create and Remove Theorems

Because all the theorems are similar only the Take rule will be examined. In List 4.2 the execution of the Take rule in PVS is shown. Because certain commands produce the same results the intervening steps have been deleted to simplify the example.

List 4.2: Execution of the Take Rule in PVS: Take_Rule_Taken: Step 1

```
Take_Rule_Taken :


  |-------
{1}   FORALL (initial: L_Graph):
        FORALL (num: nat | takerule(num, read, X, Z, initial)`PC...
            = LEND):
          takerule(num, read, X, Z, initial)`taken = TRUE


Rule? (skosimp)
```

In the List 4.2 shows what the Take rule looks like right after the prover is invoked. The first command to run is (*skosimp*) which replaces the universal quantifier FORALL (initial:L_Graph). The result of the command is in List 4.3.

List 4.3: Execution Take_Rule_Taken: Step 2

```
Take_Rule_Taken :

  |-------
{1}   FORALL (num: nat | takerule(num, read, X, Z, initial!1)‘PC =
LEND):  takerule(num, read, X, Z, initial!1)‘taken

Rule? (induct ‘‘num’’)
```

The expression in List 4.3, is of the form *(FORALL (P:pred[t]): induction subgoal implies goal)*, where p is to be instantiated by the induction predicate [SORSC99]. The command run is *(induct "num")*. Because *num* is a natural number the induction scheme is employed automatically. Employing (*induct*) simplifies the formula to a base case and induction subcases. Therefore, three subgoals result.

List 4.4: Execution Take_Rule_Taken.1

```
Take_Rule_Taken.1 :

  |-------
{1}   takerule(num!1, read, X, Z, initial!1)‘PC = LEND {2}
takerule(num!1, read, X, Z, initial!1)‘taken

Rule? (grind)
This completes the proof of Take_Rule_Taken.1.
```

The first subgoal, Take_Rule_Taken.1, is in List 4.4. The subgoal numbers are added after the theorem name. Because there is only a single consequent for this

subgoal which only requires definition expansion, *grind* is a good command to choose. *Grind* rewrites and simplifies the subgoal and proves the subgoal in this case.

The next subgoal, Take_Rule_Taken.2, is also proved using *grind*. Take_Rule_Taken.3 is another induction. After using *(induct "j")* two subgoals are generated. The first subgoal Take_Rule_Taken.3.1 is solved through *grind*. The second subgoal Take_Rule_Taken.3.2 produces 5 subgoals after "grinding".

List 4.5: Execution Take_Rule_Taken.3.2.1

```
Take_Rule_Taken.3.2.1 :
{-1}  j!1 >= 0
{-2}  L1?(takerule(j!1, read, X, Z, initial!1)`PC)
{-3}  edges(takerule(j!1, read, X, Z, initial!1)`g1)(X, Z)
{-4} takerule(j!1, read, X, Z, initial!1)`db(X, Z)(read)
  |-------

Rule? (expand ``takerule'' )
```

The first subgoal, Take_Rule_Taken.3.2.1, List 4.5 above, only has antecedents. By applying the rule *(expand "takerule")* the definition of takerule is expanded allowing the resulting expressions to be simplified using decision procedures and rewriting [SORSC99]. The resulting expression can be simplified through a grind command, which results in the List 4.6.

List 4.6: Execution Take_Rule_Taken.3.2.1

```
Take_Rule_Taken.3.2.1 :
[-1]  j!1 >= 0
{-2}  L0?(takerule(j!1 - 1, read, X, Z, initial!1)`PC)
{-3}  Y = X
{-4}  add(read, emptyset[Rights[Vertex]])(read)
  |-------
{1}   j!1 = 0
```

In Take_Rule_Taken.3.2.1 antecedent {-3} is $Y = X$. However in the definitions file the axiom *not_eq_ax* states that none of the vertices are equal to each other thus $Y! = X$. To use that axiom in the proof it must be invoked which is done through *(lemma "not_eq_ax")* command. Thus the expression in List 4.7 becomes trivial because of the contradiction of $\{-1\}$ and $[-4]$. The axiom still needs to be asserted. The (assert) command moves the antecedent $\{-1\}$ down to a consequent of the form $Y = X$ and $Z = Y$ and $X = Z$. Because $Y = X$ and $Y = X$ are in both the antecedent and the consequent, it is true, which completes the proof Take_Rule_Taken.3.2.1.

List 4.7: Execution Take_Rule_Taken.3.2.1

```
Take_Rule_Taken.3.2.1 :
{-1}  X /= Y AND Y /= Z AND Z /= X
[-2]  j!1 >= 0
[-3] L0?(takerule(j!1 - 1, read, X, Z, initial!1)‘PC)
[-4]  Y = X
[-5] add(read, emptyset[Rights[Vertex]])(read)
  |-------
[1]   j!1 = 0

Rule? (assert)
This completes the proof of Take_Rule_Taken.3.2.1.
```

Through the application (*grind*), *(expand"takerule")*, *(lemma "not_eq_ax")*, and (*assert*) the rest of the takerule theorem is proved. The entire proof is proved when "Q.E.D." is reached.

Figure 4.13 shows the complete proof schema of takerule.

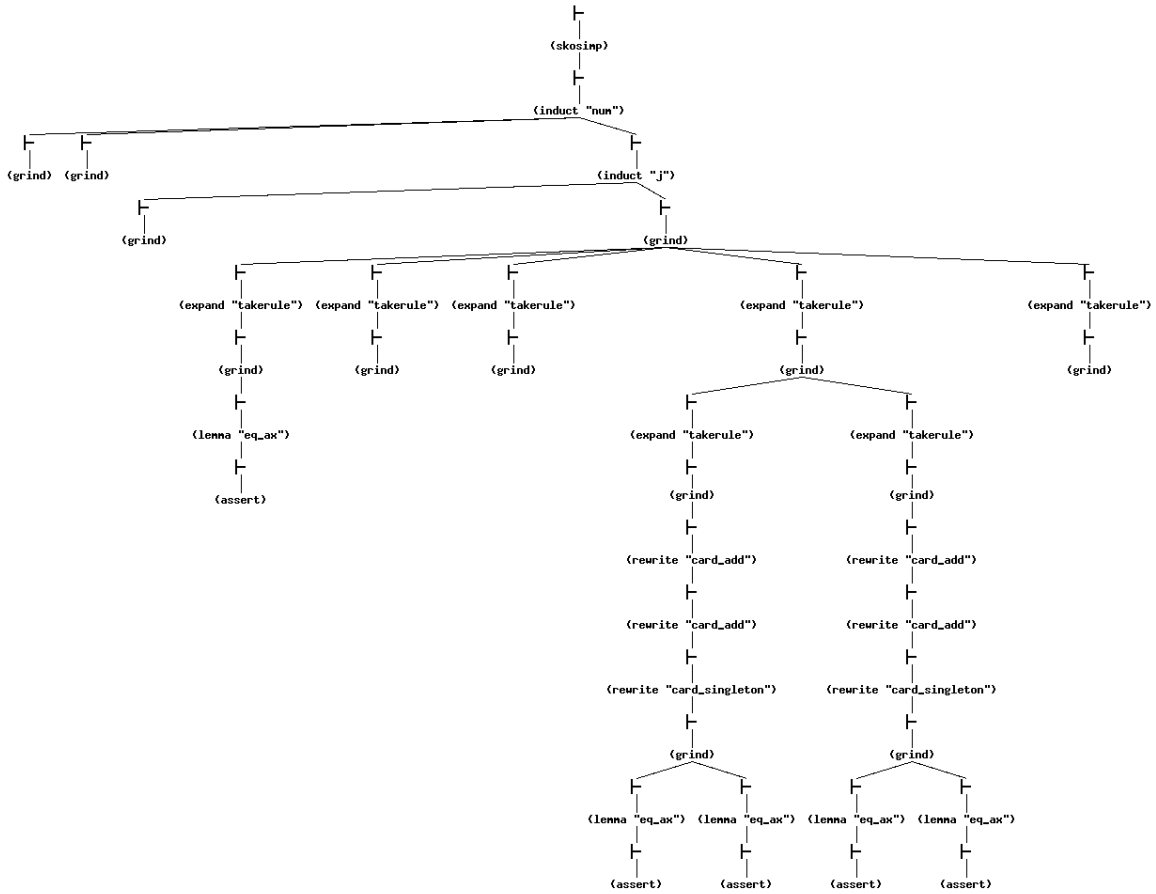

Figure 4.13: Take Picture Proof

All the rules use (*skosimp*), *(induct "num")*, *(induct "j")*, (*grind*), *(expand "theorem name")*, *(lemma "not_eq_ax")*, and (*assert*).

The application of the rules in the same order will prove the Grant rule as shown in Figure 4.14.

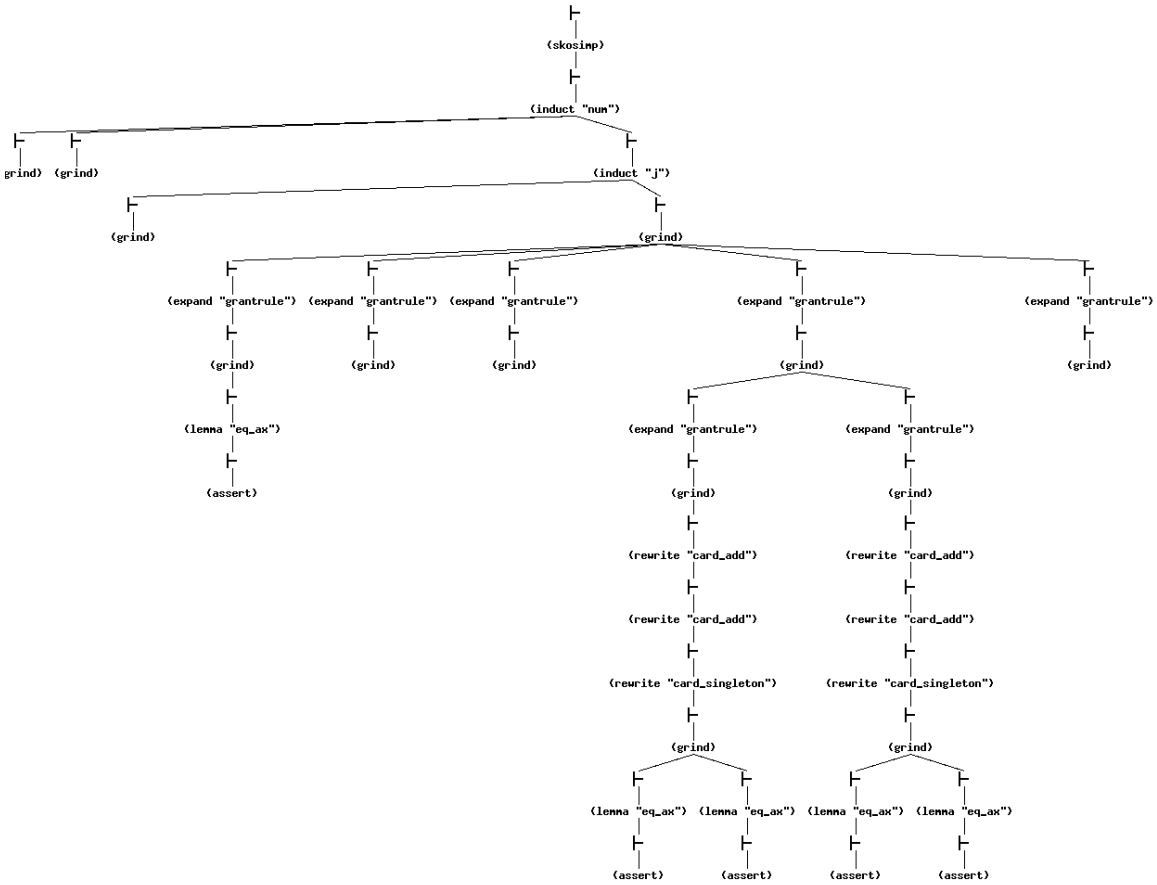

Figure 4.14: Grant Picture Proof

The Create rule, however, requires a different order of the commands applied. Because there are fewer case statements, there are fewer subgoals. The Create rule is shown in Figure 4.15.

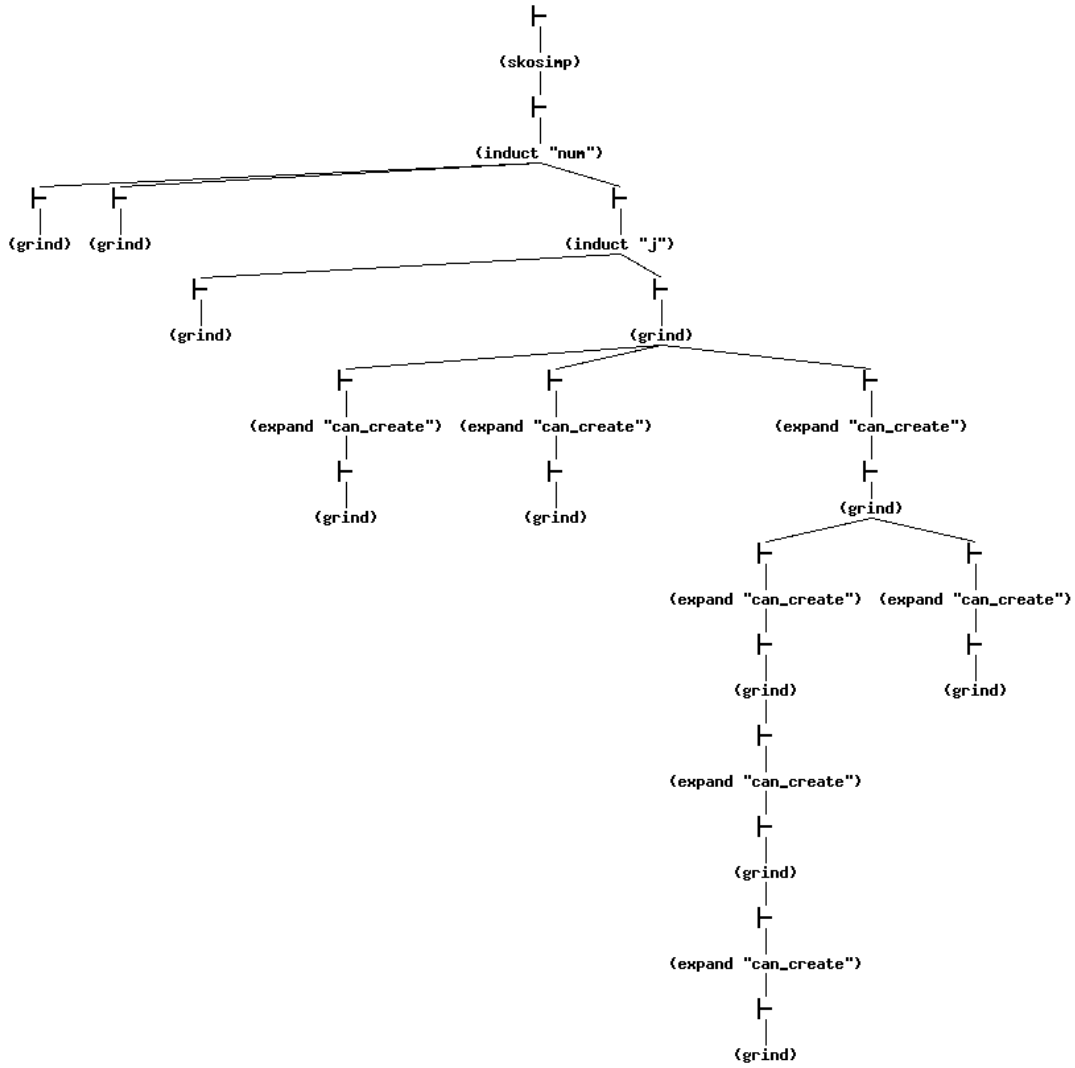

Figure 4.15: Create Picture Proof

The Remove rule also has a different number of subgoals, as Figure 4.16 shows.

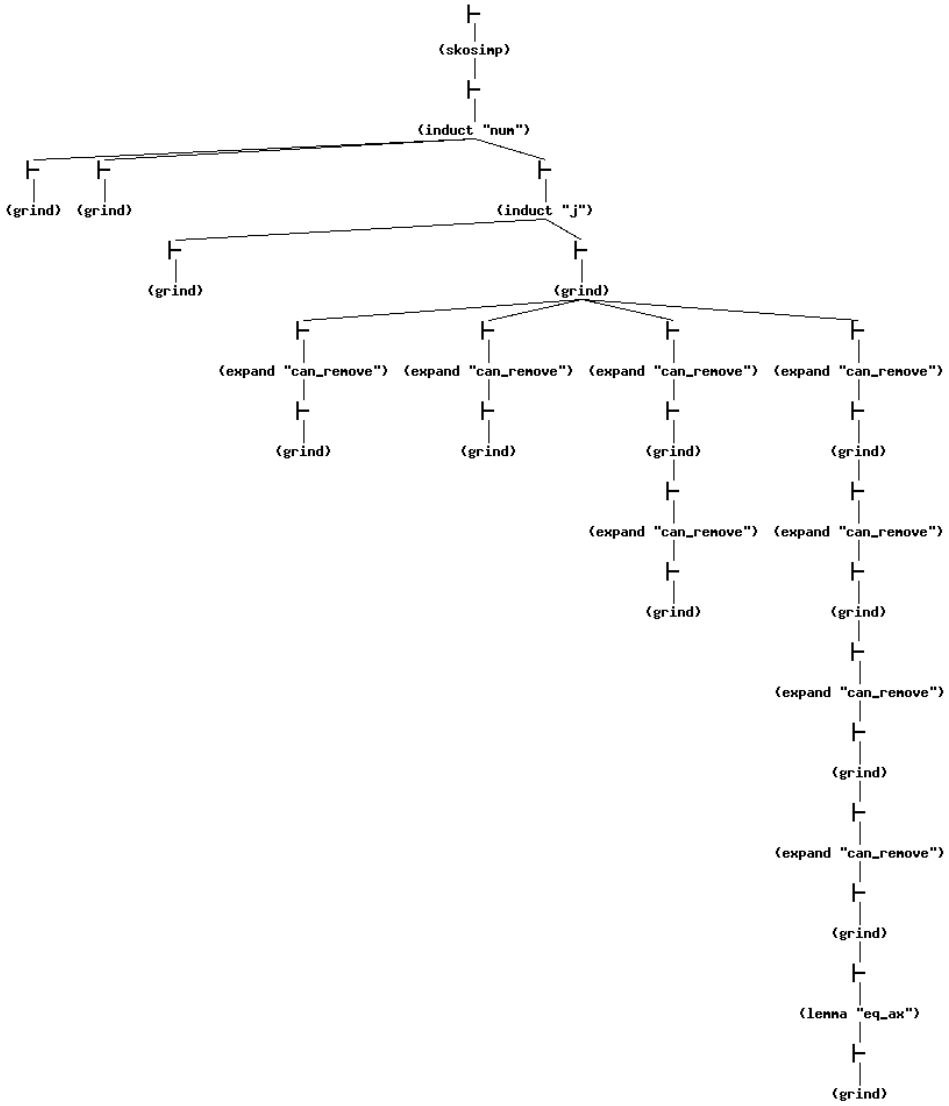

Figure 4.16: Remove Picture Proof

4.5 Can_Share Algorithm

The Can_Share algorithm is very involved and difficult to implement in PVS. It follows the same form as the Model 2 specifications. It uses recursion to move through the case statements which in turn call functions to manipulate the graph or to error check the algorithm. Ideally, no prior knowledge of the graph would be needed when the theorems are tested. After the graph is instantiated, it should not have to remembered it to get the theorems to run correctly, which is why there is error checking. For both the Take and Grant rule, three vertices and two edges are involved. This is the basis of the Can_Share algorithm and code, which was designed for a 3+ vertex graph. It finds the first three vertices and edges that correctly implement the rule then connect the outer two vertices with an edge. If there are more than 3 vertices, it will start over again. This way the problem is always working off a three node system. For simplicity, the current nodes are saved into variables labelled *one*, *two*, and *three*. In the following two subsections the imported theories are discussed then the Can_Share code.

4.5.1 Theories Imported for Can_Share. Can_Share uses two theories, Add_Edge (List 3.2) and Graph_Init (List 3.3), which are discussed above. The only difference is the definition file is changed to csDefinitions for Can_Share.

csDefinitions Theory

Except for additions to Parts 8 and 11 and the addition of a new Part 9, this theory as shown in List 4.8 is similar to the Model 2 definitions file.

List 4.8: csDefinitions Theory

```
%Part 1:
 csDefinitions  [ Vertex: Type+ ]: THEORY

BEGIN
```

```
%Part 2:
Importing digraphs@digraphs[Vertex]
Importing digraphs@digraph_ops[Vertex]
Importing digraphs@digraph_deg[Vertex]

%Part 3: %declares TYPE of right works
Rights: TYPE = {read, write, take, grant} right: TYPE =
setof[Rights] containing emptyset[Rights]

%Part 4:
E_DB: TYPE =function[edgetype[Vertex]->set[Rights]]

%Part 5:
X,Y,Z,A,B: Vertex

%Part 6:
nil: Vertex

%Part 7:
PCT: TYPE =
{L00,L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L9t,L9g,L10,L11,L12,L13,L14,L15...
   ,L16,LTRUE,LFALSE,TG_EDGE,LEND}

%Part 8
node : TYPE = [#
            dest: Vertex,
            source: Vertex,
            e1: edgetype, % outgoing selected edge
            ie: finite_set[edgetype], % all incoming edges
            e2:edgetype, %incoming selected edge
            ie1:  finite_set[edgetype], % contains incoming edges ...
               not visited
            oe: finite_set[edgetype], % all outgoing edges
            oe1: finite_set[edgetype], % edges modified
```

```
            oe2: finite_set[edgetype], % contains outgoing edges ...
               not visited
            oec: nat, % outgoing edge card
            iec:nat, % incoming edge card
            ine1:finite_set[edgetype],
            ine2:finite_set[edgetype],
            e3:edgetype, %incident selected edge
            inec:nat % incident edge card
            #]

%Part 9
n_db: TYPE = function[Vertex -> node]

%Part 10:
found:type = {yes, no, un}

%Part 11:
 L_Graph: TYPE =     [#
            g0: digraph[Vertex],
            g1: digraph[Vertex],
            db: E_DB,
            PC: PCT,
            Node: n_db,
            verts: set[Vertex],
            shared: bool,
            FOUND:set[found],
            one: Vertex,
            two: Vertex,
            three: Vertex,
            addright: Rights,
            good_nodes: set[Vertex]
            #]

%Part 12:
null: edgetype = (nil,nil)
```

```
%Part 13:
not_eq_ax: AXIOM X/=Y and Y/=Z and Z/=X and X/=A and A/=Y and A/=Z

%Part 14:
END csDefinitions
```

In Part 8, type node has more attributes outgoing edges $oe, oe1, oe2$, incoming edges $ie, ie1$, and incident edges $ine1, ine2$. Additionally nodes have attributes for a selected outgoing edge, $e1$; incoming edge, $e2$; and incident edge, $e3$. There is also a variable to track the cardinality of each edge set $oec, iec, inec$. The need for these multiple variables for edge sets will be explained with the following cNode_ops theory.

Part 9 declares a type found.

Part 11 declares a few more state variables: $g0$ is another digraph, *verts* is a set of vertices, *shared* returns true or false depending on if the right can be shared, $FOUND$ is a set of found, and the variables $one, two, three$ are vertex, and *good_nodes* is a set of vertices.

cNode_ops Theory

This theory, List 4.9 declares all the operations on the node state.

List 4.9: cNode_ops Theory

```
%Part 1:
csNode_ops [Vertex: TYPE+]: THEORY
BEGIN

%Part 2:
Importing csDefinitions[Vertex]

%Part 3:    %initialize node
Node_ops(Node: n_db, c: L_Graph, x: Vertex, oes:
```

```
 finite_set[edgetype], flag:bool, ae:edgetype):n_db = Node with
[(x):=(#
%outgoing edges
        %takes all edges
oe:= if nonempty?(oes) then oes else emptyset[edgetype]endif,
        %this should add together the difference of oe(all ...
           outgoing edges (incase one is added), and oe1
        % edges(that might be modified) with oe2 which does not ...
           contain the edges already visited.
oe1:= if nonempty?(oes) then union(difference(Node(x)`oe,
    Node(x)`oe1),Node(x)`oe2)else emptyset[edgetype]endif,
e1:=  if flag then ae else (if nonempty?(Node(x)`oe1) then
    choose(Node(x)`oe1) else null endif) endif,
        %removes edge that is visited  are going to
oe2:= if (member(Node(x)`e1,Node(x)`oe1)) and
    nonempty?(Node(x)`oe1)  then remove(Node(x)`e1,Node(x)`oe1)...
       else
    Node(x)`oe1 endif,
oec:= card(Node(x)`oe2),
%incoming edges
ie:=incoming_edges(x,c`g1),
e2:= if nonempty?(Node(x)`ie) then choose(Node(x)`ie) else null ...
   endif, %edge
ie1:=  if (member(Node(x)`e2,Node(x)`ie)) and
nonempty?(Node(x)`ie) then remove(Node(x)`e2,Node(x)`ie) else
Node(x)`ie endif, iec:=card(Node(x)`ie1),
%incident edges
ine1:= if nonempty?(incident_edges(x,c`g1)) then
    incident_edges(x,c`g1) else emptyset[edgetype] endif,
e3:= if nonempty?(Node(x)`ine1) then choose(Node(x)`ine1) else ...
   null endif, %edge
ine2:=  if (member(Node(x)`e3,Node(x)`ine1)) and
    nonempty?(Node(x)`ine1)  then remove(Node(x)`e3,Node(x)`ine1) ...
       else Node(x)`e3 endif,
inec:= card(Node(x)`ine2),
```

```
%other
source:= x,
dest:= if Node(x)‘e1=null then nil else Node(x)‘e1‘2 endif
    #)]

%Part 4:    %update node dest and source
Node_update(Node: n_db, x: Vertex, e1: Vertex, e2: Vertex):n_db =
Node  with [(x)‘dest:= e2,(x)‘source:= e1, (x)‘num:=0]

%Part 5:    %update incident  edges for node
Node_INE_update(Node: n_db, c: L_Graph, x: Vertex, ines:
finite_set[edgetype]):n_db = Node
 with [(x)‘e3:= if nonempty?(Node(x)‘ine1) then choose(Node(x)‘...
    ine1) else null endif,
    (x)‘ine2:=  if (member(Node(x)‘e3,Node(x)‘ine1)) and nonempty...
       ?(Node(x)‘ine1)
            then remove(Node(x)‘e3,Node(x)‘ine1)else Node(x)‘ine1 ...
               endif ]

%Part 6:    %initializes all nodes
set_Node(c: L_Graph, v: finite_set[Vertex], n: n_db): recursive
L_Graph  =
    if empty?(v)then   c
    else let a = choose(v) in set_Node(c, remove(a,v), Node_ops(c‘...
       Node,c,a, outgoing_edges(a, c‘g1), false, null))
endif measure  card(v)


%Part 7:  %finds the first node in all the edges that have the ...
   required right to the last node
find_good_node(c: L_Graph, e:edgetype, r:Rights): L_Graph = if not
member(r,c‘db(e)) then c else (  if nonempty?(c‘good_nodes) then c
with [good_nodes:= add(e‘1,c‘good_nodes)]
    else c with [good_nodes:=add(e‘1,emptyset[Vertex])]
    endif) endif
```

```
% send in the node  to connect to with what right  to get
% it will find all the incoming edges that has that right

%Part 8:    %finds last edge
Find_last_edge(c: L_Graph, e: finite_set[edgetype], r:Rights):
recursive L_Graph  =
    if empty?(e)then   c
    else let a = choose(e) in Find_last_edge(find_good_node(c, a, ...
       r), remove(a,e), r)
endif measure  card(e)

%Part 9:
END csNode_ops
```

In Part 3, the *Node_ops* function is the main workhorse for the theory. This function sets all the variables the first time out and is used during updates. The first variable *oe* takes all the outgoing edges for the vertex or sets itself to empty if there are none. $Oe1$ contains all the edges except for the ones already visited which allows new edges to be added. $Oe2$ contains only the edges not visited. $E1$ contains the first selected outgoing edge for the vertex. Oec is the cardinality of $oe2$. The incoming and incidental edge variables are similar to the outgoing edges. Ie contains all incoming edges. $Ie1$ is all the edges that haven't been visited. $E2$ is the selected incoming edge variable and iec is the cardinality of $ie1$. $Ine1$ contains all the incidental edges. $Ine2$ contains all the incidental edges not visited. $E3$ is the selected incidental edge. $Inec$ is the cardinality of $ine2$. *Source* is first set to the vertex itself then is set to the vertex that called it. *Dest* is the destination node that is set first set to $e1$'s second vertex.

Node_update in Part 4 updates the *source* and the *dest* for the vertex.

In Part 5, $Node\_INE\_update$ updates all the incidental edge variables.

Part 6 *set_Node* is the recursive call that originally sets each vertices' attributes.

Part 7 the *find_good_node* function takes the edge sent to it and determines if the needed right, in this example read, is in the edge database. If the right is, *good_nodes* is updated with the edges first vert (Y in this example). This will cycle through all the edges getting all the vertices that are connected to the node to share rights from. If the algorithm hits a good node the theorem will be correct.

In Part 8, *Find_last_edge* is a recursive function that goes through the last node, in this example Z, incoming edges calling the function *find_good_node* to find all the nodes that have edges connected to Z.

tgedge Theory

This theory in List 4.10 finds the tg edge for the first leg of the rule with *TG_edge*? then for the second leg of the rule with *Find_edge_out*?.

List 4.10: tgedge Theory

```
%Part 1:
tgedge [Vertex: TYPE+]  : THEORY

BEGIN
%Part 2:
Importing csDefinitions[Vertex]
Importing csNode_ops[Vertex]



%Part 3: %Don't care about Direction as long as T/G
TG_edge?(c: L_Graph, x: Vertex): L_Graph = if
member(take,c`db(c`Node(x)`e3)) or
member(grant,c`db(c`Node(x)`e3))
    % this should save x into the source node, and the start node ...
       into dest. to use Find_edge
then c with [Node:=Node_update(c`Node, x, c`Node(x)`e3`2,
c`Node(x)`e3`1), FOUND:=add(yes, emptyset[found])]
```

```
else(if c‘Node(x)‘inec =0
     then c with [FOUND:=add(no, emptyset[found])]
        % go get another incoming edge
     else c with [Node:= Node_INE_update(c‘Node, c, x, ...
        incident_edges(x, c‘g1)),  FOUND:=add(un, emptyset[found])...
         ]
     endif)endif


%Part 4:    %finds edge for the second leg of the Take rule
Find_edge_out?(c: L_Graph, x: Vertex): L_Graph =
    If  not c‘Node(x)‘oec = 0
    then c with [ Node:= Node_ops(c‘Node,c,x,outgoing_edges(x,c‘g1...
       ), false, null)]
    else c Endif

%Part 5
END tgedge
```

In Part 3 *TG_edge*? finds an incidental edge that has a take or a grant right. For the take or the grant, the direction of the edge does not matter, which is why the incidental edge is used here. If the node is found, $FOUND$ is set to "yes" and the node source and destination variables get updated. If the edge is not found and the cardinality of the edge set *inec* is not zero, then the $FOUND$ variable is set to "un", if the cardinality is zero, the set is empty and $FOUND$ is set to "no".

Part 4 *Find_edge_out*? finds the next edge for the rule. Direction is important on the second edge, because this edge might contain a read or a write. If the cardinality of the outgoing edges is zero then L_Graph is unchanged otherwise L_Graph is returned with the node updated through a call to *node_ops*

4.5.2 Can_Share Code. The following code in List 4.11 is currently non-proved, but typechecked. Can_Share is discussed in slightly different manner than the rules above because it is more involved.

List 4.11: Code for Can_Share

```
Can_Share [Vertex: TYPE+]: THEORY
%****SUBJECT only CAN_SHARE** can have a t* path or a t*g* path

BEGIN

Importing csDefinitions[Vertex] % contains the definitions
Importing csInitialization[Vertex] % initializes the graph
Importing csNode_ops[Vertex] % contains operations on the nodes
Importing csAdd_Edge[Vertex] % Adds the edges and edge rights
Importing tgedge[Vertex] % finds edges



%loop actions
    %sets: one,
L00(c: L_Graph, x:Vertex, y:Vertex): L_Graph = c with [ g0:=
InitGraph,g1:=InitGraph, db:= ADD(c`db), one:=x ]

    %sets: the node information
L2(c: L_Graph, x: Vertex):L_Graph = set_Node(c, vert(c`g1),
c`Node)

L4(c: L_Graph, x: Vertex, r:Rights):L_Graph = Find_last_edge(c,
incoming_edges(x,c`g1), r)

    %Finds the first edge from the beginning node with a t or g
TG_EDGE(c: L_Graph, x: Vertex, n: n_db):L_Graph = TG_edge?(c, x)

    %sets two
set_two(c:L_Graph, x: Vertex): L_Graph = c with [
two:=c`Node(x)`dest]
```

```
L9(c: L_Graph, x: Vertex, r:Rights):L_Graph = c with [Node:=
Node_update(c‘Node, x, c‘Node(x)‘e1‘1, c‘Node(x)‘e1‘2),
addright:=r]

    %sets: three
set_three(c:L_Graph, x: Vertex): L_Graph = c with [
three:=c‘Node(x)‘dest]

    %adds new edge to graph, adds new edge right to edge_db, and ...
        updates the node
L3(c: L_Graph, x: Vertex, y: Vertex, r: Rights):L_Graph = c with [
    g1:=AddEdge(c‘g1, x, y),
    db:= AddEdgeRight(c‘db, x, y, r),
    Node:= Node_ops(c‘Node, c, x,outgoing_edges(x, c‘g1), true, (x...
        ,y))]

    %finds edge from second node
L14(c: L_Graph, x: Vertex, n: n_db): L_Graph =Find_edge_out?(c, x)

LTRUE(c:L_Graph): L_Graph = c with [shared:= true]

LFALSE(c:L_Graph): L_Graph = c with [shared:= false]

%-----------------------------------------------------------

%Part 3:%Switch Loop
sw(c: L_Graph, r: Rights, x: Vertex,  y: Vertex): L_Graph =
    Cases c‘PC of
%Part 3.1: %initializes graph and E_DB
    L00: L00(c,x,y) with [PC:=L0],

%Part 3.2: % checks if first node is in graph else quit
    L0:c with [PC:= if not member(x, vert(g1(c))) then LFALSE
            % checks if second node is in graph else quit
```

```
        else (if not member(y, vert(g1(c))) then LFALSE
            % checks if needed edge exists else need to obtain ...
               edge
        else(if not member((x,y),outgoing_edges(x, c‘g1)) then L1
            % checks if edge exists it has right needed else need ...
               to obtainedge with correct right
        else (if member(r,(c‘db(x,y))) then LTRUE else L1 endif)
        endif)  endif)  endif],

%Part 3.3: %checks if there are at least 3 verts in graph else ...
   quit
    L1: c with [PC:= if card[Vertex](vert(g1(c))) >=3
        then L2         %yes
        else LFALSE endif], %no

%Part 3.4: %initializes the nodes and sees if there is edges going...
    into end node
    L2: L2(c, x) with [PC:=L3],

%Part 3.5: % checks to see if the node to share rights to has an ...
   incoming edge
    L3: c with [PC:=if c‘Node(y)‘iec=0 then LFALSE
        else L4 endif ],

 %Part 3.6: %goes and find all the edges going to the last node ...
    with the needed right
    L4: L4(c, x, r) with [PC:= TG_EDGE],

 %Part 3.7: %goes to find an edge with TG
    TG_EDGE:TG_EDGE(c, c‘one, c‘Node) with [PC:=L5],

%Part 3.8:
    L5: set_two(c, x) with [PC:=L6],
```

```
%Part 3.9:
    L6: c with [PC:=
        %haven't found an edge and more edges
        if member(un,c‘FOUND) then TG_EDGE
        %if not found and no more edges LFALSE else continue
        else (if member(no,c‘FOUND) then LFALSE else L7 endif) ...
           endif],

%Part 3.10: %checks if destination node has edges, then if the ...
   destination node  edge contains
        %the node needed, then if right needed is there
    L7: c with [PC:=
        %does the dest node have incidental edges
        if nonempty?(c‘Node(c‘two)‘ine2)
        then L8
        else TG_EDGE
        endif],

% is the node needed selected, then checks to see if it is the ...
   correct right for that edge
%Part 3.11:
    L8: c with [PC:=
        %checks to see if we get the node we need in X's ...
           destination nodes current edge selected
        if not member(c‘Node(c‘two)‘e3‘2, c‘good_nodes)
        %no
        then L10
        %yes: now does the dest node edge contain the right needed
        else (if (member(r, c‘db(c‘Node(c‘two)‘e3)))
        %yes
        then L9
        %no
        else L10 endif)endif],
```

```
%Part 3.12: %updates X's destination node's destination, sets the ...
   right to add to edge
    L9: L9(c, c‘two, r) with [PC:=L12],

%Part 3.13: %adds a take to edge
    L9t: L9(c, c‘two, take) with [PC:=L12],

%Part 3.14: %adds a grant to edge
    L9g: L9(c, c‘two, grant) with [PC:=L12],

%Part 3.15: %Checks if take is in the new edge set of rights
    L10: c with  [PC:=
        if (member(take, c‘db(c‘Node(c‘two)‘e3)))
        %yes
        then L9t
        %no
        else L11 endif],

%Part 3.16:        %if grant is in the edge
    L11: c with  [PC:=
        if (member(grant, c‘db(c‘Node(c‘two)‘e3)))
        %yes
        then L9g
        %no
        else L14 endif],

%Part 3.17:
    L12: set_three(c, c‘two) with [PC:=L13],

%Part 3.18: %adds edge to graph and adds right edge to database
    L13: L13(c, c‘one, c‘three, c‘addright) with [PC:=L16],

%goes to find another edge for second leg of take rule
%Part 3.19:
    L14: L14(c, c‘two, c‘Node) with [PC:= L15],
```

```
%Part 3.20:
    L15:  c with [PC:=  if nonempty?(outgoing_edges(c‘two,c‘g1))
        then L7
        else (if c‘Node(c‘one)‘oec =0
        then LFALSE
        else TG_EDGE  endif) endif],

%Part 3.21:
    L16: c with [PC:=
        % checks if edge has right needed else
        if member(r,(c‘db(x,y))) then LTRUE else TG_EDGE endif],

%Part 3.22:
    LTRUE: LTRUE(c) with [PC:= LEND],

%Part 3.23:
    LFALSE: LFALSE(c) with [PC:= LEND],

%Part 3.24:
    LEND: c

    Endcases

%Part 2:
can_share(num: nat, r: Rights, x: Vertex, y: Vertex, initial:
L_Graph ): Recursive L_Graph = if num=0 then
    initial with [PC:=L00]
else
    sw(can_share(num-1,r,x,y,initial),r,x,y)
endif measure num
```

```
%Part 1:
Can_Share_correct: THEOREM FORALL (initial: L_Graph): FORALL(num:
nat | can_share(num, read, X, Z, initial)`PC=LEND):
 can_share(num, read, X, Z, initial)`shared = true

END Can_Share
```

Part 1 is the theorem *Can_Share_correct* which calls the recursive function *Can_Share* in Part 2.

Part 2 in turn calls the function *sw*, Part 3, which is the case statements and what the recursive call from Part 2 cycles through using the program counter (PC).

Part 3.1 and 3.2 initialize the graph and do the first round of error checking. If the edge is not found with the correct right, go to Part 3.3 to try to find a path to share rights or if it is found with the correct right, go to Part 3.22 which sets shared to true.

Part 3.3 checks to see if the graph has at least 3 vertices. If it does not, go to Part 3.33 which will set shared to false. Proving there were at least 3 nodes in the graph was not accomplished.

In Part 3.4 each vertices node attributes are set. L2 calls the loop action L2 which in turn calls the *set_Nodefunction* in Node_ops. then goes to Part 3.5

Part 3.5 makes sure the node to share to, in this example Z, has incoming edges by checking the cardinality of the incoming edge variable set for Z. Another difficulty in the proof has to do with referencing the variable *iec*. If the vertex does have incoming edges denoted by the cardinality being greater than zero, go to Part 3.6 else go to Part 3.33.

Part 3.6 calls the loop action L4 which calls the function find $find\_last\_edge$ from Node_ops. This makes sure there are edges actually going into the node to share from with the set of rights needed. In this case Z has an incoming read edge. Then Part 3.7 is called.

TG_EDGE in Part 3.7 finds the first edge to enact the Can_Share rule. It calls the function $TG\_edge?$, then goes to Part 3.8.

Part 3.8 sets the *two* variable with X's *dest* value.

Part 3.9 checks the $FOUND$ variable. If "un" is the member of $FOUND$ then Part 3.7 is called again, if "no" is the member then Part 3.23 is called because shared will be false; otherwise continue to Part 3.10.

Part 3.10 checks to see if $c'two$ (X's Destination vertex) has incidental edges. If it does go to Part 3.11 else go back to Part 3.7.

Part 3.11 checks to see if $c'two$'s (Y) selected incidental edge (e3) contains a good node for the vertex to share from (Z). If it does not contain a good node go to Part 3.15. If it does contain a good node check verify again that it contains the right needed then go to Part 3.12 else go to Part 3.15.

Part 3.12 this calls the loop function L9 which calls $Node\_update$. Then Part 3.17 is called.

Part 3.13 is the same as 3.12 except the right sent to the loop action L9 is take.

Part 3.14 is the same as the Part 3.11 and 3.12 except the right sent to the Loop action L9 is grant.

Part 3.15 checks to see if edge in $c'two$'s $e3$, the selected incidental edge has a take edge. If it does Part 3.13 is called otherwise Part 3.16 is called.

Part 3.16 checks to see if edge in $c'two$'s $e3$, the selected incidental edge has a take edge. If it does, Part 3.14 is called otherwise Part 3.19 is called.

Part 3.17 this sets the $c'three$ variable. In this example it is set to Z. If the graph was bigger it would be set to a different vertex. Then, Part 3.18 is called.

Part 3.18 this calls the loop function L13 and adds the new edge to the graph and the right to the edge database. The right added is the right that was entered for the theorem. Then Part 3.17 is called.

Part 3.19 calls the loop action L14 to go find another edge, by calling the *Find_edge_out*? function from the tgedge theory. Then calls Part 3.20.

Part 3.20 checks to see if the *c'two* still has outgoing edges. If it does then Part 3.10 is called. If *c'one* (X) still has outgoing edges go to Part 3.7 else go to Part 3.23.

In Part 3.21 the newly added edge is checked to see if the right needed is part of its edge rights. If it is then Part 3.22 is called. If not, then Part 3.7 is called.

Part 3.22 is called if Can_Share is true. Then goes to Part 3.24 by setting PC equal to LEND.

Part 3.23 is called if Can_Share is false. Then goes to Part 3.24 by setting PC equal to LEND.

Part 3.24 is the end of the switch statement and signals the end of the recursion.

This code as stated above is typechecked, but yet unproved. The issues that continually occur concern the edge functions, outgoing_edges, incoming_edges, incidental_edges and the cardinality of them. Each use of node attributes appears to induce another round of recursion, which is likely the reason during the proof, the proof tree grew continually.

4.6 Summary

This chapter discusses hate PVS Prover and provides detailed examples of proof session to solve the TCCs and theorems for both model specifications, as well as Can_Share code. Chapter 5 discusses the results of this research.

V. Conclusions

5.1 Significance of Findings

The two Take-Grant rule specification models along with the Can_Share predicate automated in PVS are an important step in automating a safety proof for a given computer system. The first research goal of implementing the rules was accomplished. However, what originally seemed an easy problem, proved to be more complicated, which inspired two separate specification models.

The first model has no error checking and uses a straightforward method of incorporating the required data correctly for each theorem. Each time a function is required to prove the theorem, the graph is initialized before that function can be called. Model 1 is somewhat unwieldy and prone to human error due to the manual data entry. It is time consuming to do manual error checking since PVS typechecking and prover are of limited help if data are entered incorrectly into the functions.

Model 2 with error checking is more user friendly. Once the algorithm for each rule is correctly transformed into code, the user can test different theorems against the initialized graph. The Can_Share code allows theorems to be reused for any graph initialized.

The Take-Grant rules which originally were thought to be simple to implement in PVS turned out to be a challenge. Instead of being able to implement a complete system with subjects and objects, the rules were implemented using a subject only digraph and are too simplistic for a real computer system.

The goal of proving the consistency of formal model specifications and proving an application of the Take-Grant rules produce a valid model was partially proved. The individual Take-Grant rules did indeed produce a valid model. The power of the Take-Grant Protection Model is in the predicates, like Can_Share, which produces a better representation of a valid model.

The contributed labelled digraph theory was submitted to NASA Langley and is currently under review for inclusion in their PVS libraries.

5.2 PVS Issues

There are several issues associated with using the Prototype Verification System. PVS has a steep learning curve–it is not easily learned. It is estimated that a skilled computer scientist will require approximately six months to become proficient in PVS. Because the PVS user community is not large, it is sometimes difficult to get assistance with problems. There is a PVS help site, which provides limited help and question posting capability.

The documentation for PVS is not geared towards the beginner as it assumes familiarity with proof concepts and mechanical theorem proving structure. PVS is a logic language and not programming language. Therefore, to reason about algorithms or programs individuals must create infrastructure, namely a "state". Learning to write a formal specification is a big challenge–conceptualizing the difference between writing a mathematical specification and programming, involves learning to think "functionally."

Training classes for PVS are held every couple of years. This year a class was offered, however it occurred late in the thesis process. Discussions on implementing the Take-Grant Model during the class, indicated this research is better suited for a PhD-level effort.

One of the problems that became apparent after implementing the Model 2 specification was PVS does not support standard programming constructs such as iteration. Rather, bounded recursion is used to express an iteration, which caused several problems due to the depth of the recursive calls. Another problem is there is no visual representation of the digraph after changes are made–it all must be tested with error checking or crafting appropriate theorems.

5.3 Future Research

Future research should include completing the proof of Can_Share code, which is a key predicate for the Take-Grant Protection Model and developing a proof of the

Can_Know predicate which is key for describing the flow of information. Because the current rules do not resemble an actual system, adaptation of the current rules to use a subject/object digraph is also necessary. Adapting the model to incorporate modal logic will likely result in a more powerful and expressive model.

Appendix A. PVS Theories

To get additional proofs and/or the theory files please contact:

Air Force Institute of Technology

Dr. Rusty Baldwin

email: rusty.baldwin@afit.edu

phone: 937-255-6565 x4445

Bibliography

APN01. Stanislaw Ambroszkiewicz, Wojciech Penczek, and Tomasz Nowak. Towards Formal Specification and Verification in Cyberspace. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, pages 16–32, London, UK, 2001. Springer-Verlag.

Bis81. Matt Bishop. Hierarchical Take-Grant Protection Systems. In *SOSP '81: Proceedings of the eighth ACM symposium on operating systems principles*, pages 109–122, New York, NY, USA, 1981. ACM.

Bis95. Matt Bishop. Theft of Information in the Take-Grant Protection Model. *Journal of Computer Security*, 3(4):283–309, 1994/1995.

Bis96. Matt Bishop. Conspiracy and Information Flow in the Take-Grant Protection Model. *Journal of Computer Security*, 4(4):331–359, 1996.

Bis03. Matt Bishop. *Computer Security: Art and Science*. Boston, MA : Addison-Wesley, 2003.

BM07. R. Baldwin and B. Mullins. Modeling and Analysis of Cyber Security, Tracking, and Targeting using Modal Logics. January 2007.

BS79. Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 45–54, New York, NY, USA, 1979. ACM.

CM06. J. Cheng and J. Miura. Deontic Relevant Logic as the Logical Basis for Specifying, Verifying, And Reasoning About Information Security and Information Assurance, 2006.

CS96. F. Cuppens and C. Saurel. Specifying a Security Policy: a Case Study. In *CSFW '96: Proceedings of the 9th IEEE workshop on Computer Security Foundations*, page 123, Washington, DC, USA, 1996. IEEE Computer Society.

FB96. J. Frank and M. Bishop. Extending the Take-Grant Protection System. Technical report, Computer Security Laboratory, University of California Davis, December 1996.

For03. Formal Methods Program. Formal Methods Roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003.

Gar07. James Garson. *Modal Logic*. The Stanford Encyclopedia of Philosophy. Summer 2007.

HC96. G. David Hughes and M. J. Cresswell. *A new introduction to modal logic*. Routledge, London, New York, 1996.

HR04. Michael Huth and Mark Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, Cambridge, 2004.

JMS06. G. Baramidze A. Sheth J. Miller, P. Fishwick and G. Silver. Ontologies for Modeling and Simulation: An Extensible Framework. Technical Report Technical Report No. UGA-CS-LSDIS-TR-06-011, University of Georgia, Athens, Department of Computer Science, University of Georgia, Athens, Georgia, 2006.

Kol02. G. Kolaczek. Application of Deontic Logic in Role-Based Access Control. *International Journal of Applied Mathematics and Computer Science*, 12(2), 2002.

LM82. A. Lockman and N. Minsky. Unidirectional Transport of Rights and TakeGrant Control. *Software Engineering, IEEE Transactions on*, SE-8(6):597–604, 1982.

LZ97. J. Leiwo and Y. Zheng. A Formal Model to Aid Documenting and Harmonizing of Information Security Requirements. In *SEC'97: Proceedings of the IFIP TC11 13 international conference on Information Security (SEC '97) on Information security in research and business*, pages 25–38, London, UK, UK, 1997. Chapman & Hall, Ltd.

Mar93. D. Marc. A Petri net representation of the Take-Grant model. *Proceedings Computer Security Foundations Workshop VI, 1993* , pages 99–108, 15-17 Jun 1993.

Min78. N. Minsky. An Operation-Control Scheme for Authorization in Computer Systems. *International Journal of Computing and Information Sciences*, June 1978.

Nat96. National Institute Of Standards and Technology. An Introduction to Computer Security: The NIST Handbook. Technical Report Special Publication 800-12, National Institute of Standards and Technology, Technology Administration,U.S. Department of Commerce, 1996.

Oak08. Oak Ridge National Laboratory. Oak Ridge National Laboratory - Overviews: http://www.ioc.ornl.gov/overviews.shtml, accessed 29 Jan 2008. website, Jan 2008.

OS03. Sam Owre and Natarajan Shankar. The PVS Prelude Library. Technical Report SRI-CSL-03-01, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2003.

OSRSC99a. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

OSRSC99b. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

Pad90. L.J. La Padula. Formal modeling in a generalized framework for access control. *Proceedings Computer Security Foundations Workshop III, 1990.* , pages 100–109, 12-14 Jun 1990.

SM93. P. Syverson and C. Meadows. A logical language for specifying cryptographic protocol requirements. *Proceedings. 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177, 24-26 May 1993.

SM03. A. Sabelfeld and A.C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

Sny81. Lawrence Snyder. Theft And Conspiracy In The Take-Grant Protection Model. *Journal of Computer and System Sciences*, 23(3):333–347, 1981.

SORSC99. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

SRI08. SRI International. Formalware at SRI Website. 2008.

TL04. Mahesh V. Tripunitara and Ninghui Li. Comparing the Expressive Power of Access Control Models. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 62–71, New York, NY, USA, 2004. ACM Press.

Wan05. Andy Ju An Wang. Information Security Models and Metrics. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 178–184, New York, NY, USA, 2005. ACM Press.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2008	Master's Thesis	May 2006 — Mar 2008

4. TITLE AND SUBTITLE

Applying Automated Theorem Proving to Computer Security

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Kelly McElroy, Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/08-16

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Steve Rogers
AFRL/RY
2241 Avionics Circle
WPAFB, OH 45434
674–9891
steven.rogers@wpafb.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

While more and more data is stored and accessed electronically, better access control methods need to be implemented for computer security. Formal modelling and analysis have been successfully used in certain areas of computer systems, such as verifying the security properties of cryptographic and authentication protocols. However, formal models for computer systems in cyberspace, like networks, have hardly advanced. A highly regarded graduate textbook cites the Take-Grant model created in 1977 as one of the "current" examples of security modelling and analysis techniques. This model is rarely used in practice though. This research implements the Take-Grant Protection model's four de jure rules and Can_Share predicate in the Prototype Verification System (PVS) which automates model checking and theorem proving. This facilitates the ability to test a given Take-Grant model against many systems which are modelled using digraphs. Two models, one with error checking and one without, are created to implement take-grant rules. The first model that does not have error checking incorporated requires manual error checking. The second model uses recursion to allow for the error checking. The Can_Share theorem requires further development.

15. SUBJECT TERMS

Take-Grant Protection Model, Prototype Verification Model (PVS)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Rusty O. Baldwin
U	U	U	UU	117	19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636, ext 4445, rusty.baldwin@afit.edu